Cryptography: An Introduction (3rd Edition)

Nigel Smart

Preface To Third Edition

The third edition contains a number of new chapters, and various material has been moved around.

- The chapter on Stream Ciphers has been split into two. One chapter now deals with the general background and historical matters, the second chapter deals with modern constructions based on LFSR's. The reason for this is to accomodate a major new section on the Lorenz cipher and how it was broken. This compliments the earlier section on the breaking of the Enigma machine. I have also added a brief discussion of the A5/1 cipher, and added some more diagrams to the discussion on modern stream ciphers.
- I have added CTR mode into the discussion on modes of operation for block ciphers. This is because CTR mode is becoming more used, both by itself and as part of more complex modes which perform full authenticated encryption. Thus it is important that students are exposed to this mode.
- I have reordered various chapters and introduced a new part on protocols, in which we cover secret sharing, oblvious transfer and multi-party computation. This compliments the topics from the previous edition of commitment schemes and zero-knowledge protocols, which are retained a moved around a bit. Thus the second edition's Part 3 has now been split into two parts, the material on zero-knowledge proofs has now been moved to Part 5 and this has been extended to include other topics, such as oblivious transfer and secure multi-party computation.
- The new chapter on secret sharing contains a complete description of how to recombine shares in the Shamir secret-sharing method in the presence of malicious adversaries. To our knowledge this is not presented in any other elementary textbook, although it does occur in some lecture notes available on the internet. We also present an overview of Shoup's method for obtaining threshold RSA signatures.
- A small section detailing the linkage between zero-knowledge and the complexity class $\mathcal{NP}$ has been added.

The reason for including extra sections etc, is that we use this text in our courses at Bristol, and so when we update our lecture notes I also update these notes. In addition at various points students do projects with us, a number of recent projects have been on multi-party computation and hence these students have found a set of notes useful in starting their projects. We have also introduced a history of computing unit in which I give a few lectures on the work at Bletchley.

Special thanks for aspects of the third edition go to Dan Bernstein and Ivan Damgård, who were patient in explaining a number of issues to me for inclusion in the new sections. Also thanks to Endre Bangerter, Jiun-Ming Chen, Thomas Johansson, Parimal Kumar, David Rankin, Berry Schoenmakers and Steve Williams for providing comments, spotting typos and feedback on earlier drafts and versions.

The preface to the second edition follows:

Preface To Second Edition

The first edition of this book was published by McGraw-Hill. They did not sell enough to warrant a second edition, mainly because they did not think it worth while to allow people in North America to buy it. Hence, the copyright has returned to me and so I am making it available for free via the web.

In this second edition I have taken the opportunity to correct the errors in the first edition, a number of which were introduced by the typesetters. I have also used a more pleasing font to the eye (so for example a y in a displayed equation no longer looks somewhat like a Greek letter γ). I have also removed parts which I was not really happy with, hence out have gone all exercises and Java examples etc.

I have also extended and moved around a large amount of topics. The major changes are detailed below:

- The section on the Enigma machine has been extended to a full chapter.
- The material on hash functions and message authentication codes has now been placed in a seperate chapter and extended somewhat.
- The material on stream ciphers has also been extracted into a seperate chapter and been slightly extended, mainly with more examples.
- The sections on zero-knowledge proofs have been expanded and more examples have been added. The previous treatment was slightly uneven and so now a set of examples of increasing difficulty are introduced until one gets to the protocol needed in the voting scheme which follows.
- A new chapter on the KEM/DEM method of constructing hybrid ciphers. The chapter discusses RSA-KEM and the discussion on DHIES has been moved here and now uses the Gap-Diffie–Hellman assumption rather than the weird assumption used in the original.
- Minor notational updates are as follows: Permutations are now composed left to right, i.e. they operate on elements "from the right". This makes certain things in the sections on the Enigma machine easier on the eye.

One may ask why does one need yet another book on cryptography? There are already plenty of books which either give a rapid introduction to all areas, like that of Schneier, or one which gives an encyclopedic overview, like the *Handbook of Applied Cryptography* (hereafter called *HAC*). However, neither of these books is suitable for an undergraduate course. In addition, the approach to engineering public key algorithms has changed remarkably over the last few years, with the advent of 'provable security'. No longer does a cryptographer informally argue why his new algorithm is secure, there is now a framework within which one can demonstrate the security relative to other well-studied notions.

Cryptography courses are now taught at all major universities, sometimes these are taught in the context of a Mathematics degree, sometimes in the context of a Computer Science degree and sometimes in the context of an Electrical Engineering degree. Indeed, a single course often needs to meet the requirements of all three types of students, plus maybe some from other subjects who are taking the course as an 'open unit'. The backgrounds and needs of these students are different, some will require a quick overview of the current algorithms in use, whilst others will want an introduction to the current research directions. Hence, there seems to be a need for a textbook

which starts from a low level and builds confidence in students until they are able to read, for example *HAC* without any problems.

The background I assume is what one could expect of a third or fourth year undergraduate in computer science. One can assume that such students have met the basics of discrete mathematics (modular arithmetic) and a little probability before. In addition, they would have at some point done (but probably forgotten) elementary calculus. Not that one needs calculus for cryptography, but the ability to happily deal with equations and symbols is certainly helpful. Apart from that I introduce everything needed from scratch. For those students who wish to dig into the mathematics a little more, or who need some further reading, I have provided an appendix (Appendix A) which covers most of the basic algebra and notation needed to cope with modern public key cryptosystems.

It is quite common for computer science courses not to include much of complexity theory or formal methods. Many such courses are based more on software engineering and applications of computer science to areas such as graphics, vision or artificial intelligence. The main goal of such courses is in training students for the workplace rather than delving into the theoretical aspects of the subject. Hence, I have introduced what parts of theoretical computer science I need, as and when required. One chapter is therefore dedicated to the application of complexity theory in cryptography and one deals with formal approaches to protocol design. Both of these chapters can be read without having met complexity theory or formal methods before.

Much of the approach of the book in relation to public key algorithms is reductionist in nature. This is the modern approach to protocol design and this differentiates the book from other treatments. This reductionist approach is derived from techniques used in complexity theory, where one shows that one problem reduces to another. This is done by assuming an oracle for the second problem and showing how this can be used to solve the first. At many places in the book cryptographic schemes are examined from this reductionist approach and at the end I provide a quick overview of provable security.

I am not mathematically rigorous at all steps, given the target audience, but aim to give a flavour of the mathematics involved. For example I often only give proof outlines, or may not worry about the success probabilities of many of our reductions. I try to give enough of the gory details to demonstrate why a protocol has been designed in a certain way. Readers wishing a more in-depth study of the various points covered or a more mathematically rigorous coverage should consult one of the textbooks or papers in the Further Reading sections at the end of each chapter.

On the other hand we use the terminology of groups and finite fields from the outset. This is for two reasons. Firstly, it equips students with the vocabulary to read the latest research papers, and hence enables students to carry on their studies at the research level. Secondly, students who do not progress to study cryptography at the postgraduate level will find that to understand practical issues in the 'real world', such as API descriptions and standards documents, a knowledge of this terminology is crucial. We have taken this approach with our students in Bristol, who do not have any prior exposure to this form of mathematics, and find that it works well as long as abstract terminology is introduced alongside real-world concrete examples and motivation.

I have always found that when reading protocols and systems for the first time the hardest part is to work out what is public information and which information one is trying to keep private. This is particularly true when one meets a public key encryption algorithm for the first time, or one is deciphering a substitution cipher. I have hence introduced a little colour coding into the book, generally speaking items in red are secret and should never be divulged to anyone. Items in blue are public information and are known to everyone, or are known to the party one is currently pretending to be.

For example, suppose one is trying to break a system and recover some secret message m; suppose the attacker computes some quantity b. Here the red refers to the quantity the attacker

does not know and blue refers to the quantity the attacker does know. If one is then able to write down, after some algebra,

$$b = \cdots = m,$$

then it is clear something is wrong with our cryptosystem. The attacker has found out something he should not.

This colour coding will be used at all places where it adds something to the discussion. In other situations, where the context is clear or all data is meant to be secret, I do not bother with the colours.

To aid self-study each chapter is structured as follows:

- A list of items the chapter will cover, so you know what you will be told about.
- The actual chapter contents.
- A summary of what the chapter contains. This will be in the form of revision notes, if you wish to commit anything to memory it should be these facts.
- Further Reading. Each chapter contains a list of a few books or papers from which further information could be obtained. Such pointers are mainly to material which you should be able to tackle given that you have read the prior chapter. Since further information on almost any topic in cryptography can be obtained from reading *HAC* I do not include a pointer to *HAC* in any chapter. It is left, as a general recommendation to the reader, to follow up any topic in further detail by reading what *HAC* has to say.

There are no references made to other work in this book, it is a textbook and I did not want to break the flow with references to this, that and the other. Therefore, you should not assume that ANY of the results in this book are my own, in fact NONE are my own. For those who wish to obtain pointers to the literature, you should consult one of the books mentioned in the Further Reading sections, or you should consult *HAC*.

The book is divided into four parts. Part 1 gives the mathematical background needed and could be skipped at first reading and referred back to when needed. Part 2 discusses symmetric key encryption algorithms and the key distribution problem that results from their use. Part 3 discusses public key algorithms for encryption and signatures and some additional key concepts such as certificates, commitment schemes and zero-knowledge proofs. Part 5 is the most advanced section and covers a number of issues at the more theoretical end of cryptography, including the modern notion of provable security. Our presentation of the public key algorithms in Part 3 has been designed as a gentle introduction to some of the key concepts in Part 5. Part 5 should be considered a gentle, and non-rigorous, introduction to theoretical aspects of modern cryptography.

For those instructors who wish to give a rapid introduction to modern cryptography, in a 20–30 lecture course, I recommend Chapters 3, 7, 8, 10, 11, 14 and 16 with enough of Chapter 1 so as to enable the students to understand the following material. For those instructors wishing to use this book to give a grounding in the mathematics required for modern public key cryptography (for example a course aimed at Math Majors) then I suggest covering Chapters 3, 11, 12, 13 and 15. Instructors teaching an audience of Computer Scientists are probably advised to skip Chapters 2, 12 and 13, since these chapters are more mathematical in nature.

I would like to thank the students at Bristol who have commented on both our courses, the original a draft of this book and the first edition. In addition the following people have helped me by providing detailed feedback on a variety of chapters and topics, plus they have also helped find errors: Nils Anderson, Ian Blake, Colin Boyd, Reza Rezaeian Farashahi, Florian Hess, Nick

Howgrave-Graham, Ellen Jochemsz, Bruce McIntosh, John Malone-Lee, Wenbo Mao, John Merriman, Phong Nguyen, Dan Page, Christopher Peikert, Vincent Rijmen, Ron Rivest, Edlyn Teske and Frederik Vercauteren.

Nigel Smart
University of Bristol

Further Reading

A.J. Menezes, P. van Oorschot and S.A. Vanstone. *The Handbook of Applied Cryptography.* CRC Press, 1997.

Contents

Preface To Third Edition 3

Preface To Second Edition 5

Part 1. Mathematical Background 13

Chapter 1. Modular Arithmetic, Groups, Finite Fields and Probability 3
1. Modular Arithmetic 3
2. Finite Fields 7
3. Basic Algorithms 10
4. Probability 19

Chapter 2. Elliptic Curves 23
1. Introduction 23
2. The Group Law 25
3. Elliptic Curves over Finite Fields 28
4. Projective Coordinates 31
5. Point Compression 32

Part 2. Symmetric Encryption 35

Chapter 3. Historical Ciphers 37
1. Introduction 37
2. Shift Cipher 39
3. Substitution Cipher 41
4. Vigenère Cipher 44
5. A Permutation Cipher 47

Chapter 4. The Enigma Machine 49
1. Introduction 49
2. An Equation For The Enigma 51
3. Determining The Plugboard Given The Rotor Settings 53
4. Double Encryption Of Message Keys 56
5. Determining The Internal Rotor Wirings 57
6. Determining The Day Settings 61
7. The Germans Make It Harder 62
8. Known Plaintext Attack And The Bombe's 64
9. Ciphertext Only Attack 72

Chapter 5. Information Theoretic Security 77
1. Introduction 77
2. Probability and Ciphers 78
3. Entropy 83

4. Spurious Keys and Unicity Distance 88

Chapter 6. Historical Stream Ciphers 93
1. Introduction To Symmetric Ciphers 93
2. Stream Cipher Basics 95
3. The Lorenz Cipher 96

Chapter 7. Modern Stream Ciphers 109
1. Linear Feedback Shift Registers 109
2. Combining LFSRs 115
3. RC4 119

Chapter 8. Block Ciphers 123
1. Introduction To Block Ciphers 123
2. Feistel Ciphers and DES 125
3. Rijndael 131
4. Modes of Operation 134

Chapter 9. Symmetric Key Distribution 141
1. Key Management 141
2. Secret Key Distribution 143
3. Formal Approaches to Protocol Checking 148

Chapter 10. Hash Functions and Message Authentication Codes 153
1. Introduction 153
2. Hash Functions 153
3. Designing Hash Functions 155
4. Message Authentication Codes 159

Part 3. Public Key Encryption and Signatures 165

Chapter 11. Basic Public Key Encryption Algorithms 167
1. Public Key Cryptography 167
2. Candidate One-way Functions 168
3. RSA 172
4. ElGamal Encryption 178
5. Rabin Encryption 180

Chapter 12. Primality Testing and Factoring 183
1. Prime Numbers 183
2. Factoring Algorithms 187
3. Modern Factoring Methods 192
4. Number Field Sieve 194

Chapter 13. Discrete Logarithms 201
1. Introduction 201
2. Pohlig–Hellman 201
3. Baby-Step/Giant-Step Method 204
4. Pollard Type Methods 206
5. Sub-exponential Methods for Finite Fields 212
6. Special Methods for Elliptic Curves 213

Chapter 14. Key Exchange and Signature Schemes 217

1. Diffie–Hellman Key Exchange 217
2. Digital Signature Schemes 219
3. The Use of Hash Functions In Signature Schemes 221
4. The Digital Signature Algorithm 222
5. Schnorr Signatures 226
6. Nyberg–Rueppel Signatures 228
7. Authenticated Key Agreement 229

Chapter 15. Implementation Issues 233
1. Introduction 233
2. Exponentiation Algorithms 233
3. Exponentiation in RSA 238
4. Exponentiation in DSA 238
5. Multi-precision Arithmetic 239
6. Finite Field Arithmetic 245

Chapter 16. Obtaining Authentic Public Keys 255
1. Generalities on Digital Signatures 255
2. Digital Certificates and PKI 256
3. Example Applications of PKI 259
4. Other Applications of Trusted Third Parties 263
5. Implicit Certificates 264
6. Identity Based Cryptography 265

Part 4. Security Issues 269

Chapter 17. Attacks on Public Key Schemes 271
1. Introduction 271
2. Wiener's Attack on RSA 271
3. Lattices and Lattice Reduction 273
4. Lattice Based Attacks on RSA 277
5. Partial Key Exposure Attacks 282
6. Fault Analysis 283

Chapter 18. Definitions of Security 287
1. Security of Encryption 287
2. Security of Actual Encryption Algorithms 291
3. A Semantically Secure System 294
4. Security of Signatures 296

Chapter 19. Complexity Theoretic Approaches 299
1. Polynomial Complexity Classes 299
2. Knapsack-Based Cryptosystems 302
3. Bit Security 306
4. Random Self-reductions 308
5. Randomized Algorithms 309

Chapter 20. Provable Security: With Random Oracles 313
1. Introduction 313
2. Security of Signature Algorithms 315
3. Security of Encryption Algorithms 320

Chapter 21. Hybrid Encryption 327

1. Introduction 327
2. Security of Symmetric Ciphers 327
3. Hybrid Ciphers 330
4. Constructing KEMs 331

Chapter 22. Provable Security: Without Random Oracles 337
1. Introduction 337
2. The Strong RSA Assumption 337
3. Signature Schemes 338
4. Encryption Algorithms 340

Part 5. Advanced Protocols 345

Chapter 23. Secret Sharing Schemes 347
1. Introduction 347
2. Access Structures 347
3. General Secret Sharing 349
4. Reed–Solomon Codes 351
5. Shamir Secret Sharing 356
6. Application: Shared RSA Signature Generation 358

Chapter 24. Commitments and Oblivious Transfer 361
1. Introduction 361
2. Commitment Schemes 361
3. Oblivious Transfer 365

Chapter 25. Zero-Knowledge Proofs 369
1. Showing a Graph Isomorphism in Zero-Knowledge 369
2. Zero-Knowledge and $\mathcal{NP}$ 371
3. Sigma Protocols 372
4. An Electronic Voting System 378

Chapter 26. Secure Multi-Party Computation 383
1. Introduction 383
2. The Two-Party Case 384
3. The Multi-Party Case: Honest-but-Curious Adversaries 388
4. The Multi-Party Case: Malicious Adversaries 392

Appendix A. Basic Mathematical Terminology 395
1. Sets 395
2. Relations 395
3. Functions 397
4. Permutations 398
5. Operations 400
6. Groups 402
7. Rings 410
8. Fields 411
9. Vector Spaces 412

Appendix. Index 417

Part 1

Mathematical Background

Before we tackle cryptography we need to cover some basic facts from mathematics. Much of the following can be found in a number of university 'Discrete Mathematics' courses aimed at Computer Science or Engineering students, hence one hopes not all of this section is new. For those who want more formal definitions of concepts, there is Appendix A at the end of the book.

This part is mainly a quick overview to allow you to start on the main book proper, hence you may want to first start on Part 2 and return to Part 1 when you meet some concept you are not familiar with.

CHAPTER 1

Modular Arithmetic, Groups, Finite Fields and Probability

Chapter Goals

- To understand modular arithmetic.
- To become acquainted with groups and finite fields.
- To learn about basic techniques such as Euclid's algorithm, the Chinese Remainder Theorem and Legendre symbols.
- To recap on basic ideas from probability theory.

1. Modular Arithmetic

Much of this book will be spent looking at the applications of modular arithmetic, since it is fundamental to modern cryptography and public key cryptosystems in particular. Hence, in this chapter we introduce the basic concepts and techniques we shall require.

The idea of modular arithmetic is essentially very simple and is identical to the 'clock arithmetic' you learn in school. For example, converting between the 24-hour and the 12-hour clock systems is easy. One takes the value in the 24-hour clock system and reduces the hour by 12. For example 13 : 00 in the 24-hour clock system is one o'clock in the 12-hour clock system, since 13 modulo 12 is equal to one.

More formally, we fix a positive integer N which we call the *modulus*. For two integers a and b we write $a = b \pmod N$ if N divides $b - a$, and we say that a and b are *congruent* modulo N. Often we are lazy and just write $a = b$, if it is clear we are working modulo N.

We can also consider `(mod N)` as a postfix operator on an integer which returns the smallest non-negative value equal to the argument modulo N. For example

$$18 \pmod 7 = 4,$$
$$-18 \pmod 7 = 3.$$

The modulo operator is like the C operator %, except that in this book we usually take representatives which are non-negative. For example in C or Java we have,

```
(-3)%2 = -1
```

whilst we shall assume that $(-3) \pmod 2 = 1$.

For convenience we define the set:

$$\mathbb{Z}/N\mathbb{Z} = \{0, \ldots, N-1\}$$

which is the set of remainders modulo N. This is the set of values produced by the postfix operator `(mod N)`. Note, some authors use the alternative notation of $\mathbb{Z}_N$ for the set $\mathbb{Z}/N\mathbb{Z}$, however, in this book we shall stick to $\mathbb{Z}/N\mathbb{Z}$.

The set $\mathbb{Z}/N\mathbb{Z}$ has two basic operations on it, namely addition and multiplication. These are defined in the obvious ways, for example:

$$(11 + 13) \pmod{16} = 24 \pmod{16} = 8$$

since $24 = 1 \cdot 16 + 8$ and

$$(11 \cdot 13) \pmod{16} = 143 \pmod{16} = 15$$

since $143 = 8 \cdot 16 + 15$.

1.1. Groups and Rings. Addition and multiplication modulo N work almost the same as arithmetic over the reals or the integers. In particular we have the following properties:

(1) Addition is closed :
$$\forall a, b \in \mathbb{Z}/N\mathbb{Z} : \ a + b \in \mathbb{Z}/N\mathbb{Z}.$$
(2) Addition is associative :
$$\forall a, b, c \in \mathbb{Z}/N\mathbb{Z} : \ (a + b) + c = a + (b + c).$$
(3) 0 is an additive identity :
$$\forall a \in \mathbb{Z}/N\mathbb{Z} : \ a + 0 = 0 + a = a.$$
(4) The additive inverse always exists :
$$\forall a \in \mathbb{Z}/N\mathbb{Z} : \ a + (N - a) = (N - a) + a = 0.$$
(5) Addition is commutative :
$$\forall a, b \in \mathbb{Z}/N\mathbb{Z} : \ a + b = b + a.$$
(6) Multiplication is closed :
$$\forall a, b \in \mathbb{Z}/N\mathbb{Z} : \ a \cdot b \in \mathbb{Z}/N\mathbb{Z}.$$
(7) Multiplication is associative :
$$\forall a, b, c \in \mathbb{Z}/N\mathbb{Z} : \ (a \cdot b) \cdot c = a \cdot (b \cdot c).$$
(8) 1 is a multiplicative identity :
$$\forall a \in \mathbb{Z}/N\mathbb{Z} : \ a \cdot 1 = 1 \cdot a = a.$$
(9) Multiplication and addition satisfy the distributive law :
$$\forall a, b, c \in \mathbb{Z}/N\mathbb{Z} : \ (a + b) \cdot c = a \cdot c + b \cdot c.$$
(10) Multiplication is commutative :
$$\forall a, b \in \mathbb{Z}/N\mathbb{Z} : \ a \cdot b = b \cdot a.$$

Many of the sets we will encounter have a number of these properties, so we give special names to these sets as a shorthand.

Definition 1.1 (Groups). *A group is a set with an operation which*

- *is closed,*
- *has an identity,*
- *is associative,*
- *every element has an inverse.*

A group which is commutative is often called *abelian*. Almost all groups that one meets in cryptography are abelian, since the commutative property is what makes them cryptographically interesting. Hence, any set with properties 1, 2, 3 and 4 above is called a group, whilst a set with properties 1, 2, 3, 4 and 5 is called an abelian group.

Standard examples of groups which one meets all the time at high school are:

- The integer, real or complex numbers under addition. Here the identity is 0 and the inverse of x is $-x$, since $x + (-x) = 0$.
- The non-zero rational, real or complex numbers under multiplication. Here the identity is 1 and the inverse of x is x^{-1}, since $x \cdot x^{-1} = 1$.

A group is called *multiplicative* if we tend to write its group operation in the same way as one does for multiplication, i.e.

$$f = g \cdot h \text{ and } g^5 = g \cdot g \cdot g \cdot g \cdot g.$$

We use the notation $(G, \cdot)$ in this case if there is some ambiguity as to which operation on G we are considering. A group is called *additive* if we tend to write its group operation in the same way as one does for addition, i.e.

$$f = g + h \text{ and } 5 \cdot g = g + g + g + g + g.$$

In this case we use the notation $(G, +)$ if there is some ambiguity. An abelian group is called *cyclic* if there is a special element, called the *generator*, from which every other element can be obtained either by repeated application of the group operation, or by the use of the inverse operation. For example, in the integers under addition every positive integer can be obtained by repeated addition of 1 to itself, e.g. 7 can be expressed by

$$7 = 1 + 1 + 1 + 1 + 1 + 1 + 1.$$

Every negative integer can be obtained from a positive integer by application of the additive inverse operator, $x \to -x$. Hence, we have that 1 is a generator of the integers under addition.

If g is a generator of the cyclic group G we often write $G = \langle g \rangle$. If G is multiplicative then every element h of G can be written as

$$h = g^x,$$

whilst if G is additive then every element h of G can be written as

$$h = x \cdot g,$$

where x in both cases is some integer called the discrete logarithm of h to the base g.

As well as groups we also define the concept of a ring.

DEFINITION 1.2 (Rings). *A ring is a set with two operations, usually denoted by $+$ and $\cdot$ for addition and multiplication, which satisfies properties 1 to 9 above. We can denote a ring and its two operations by the triple $(R, \cdot, +)$.*

If it also happens that multiplication is commutative we say that the ring is commutative.

This may seem complicated but it sums up the type of sets one deals with all the time, for example the infinite commutative rings of integers, real or complex numbers. In fact in cryptography things are even easier since we only need to consider finite rings, like the commutative ring of integers modulo N, $\mathbb{Z}/N\mathbb{Z}$.

1.2. Euler's ϕ Function. In modular arithmetic it will be important to know when, given a and b, the equation

$$a \cdot x = b \pmod{N}$$

has a solution. For example there is exactly one solution to the equation

$$7x = 3 \pmod{143},$$

but there are no solutions to the equation

$$11x = 3 \pmod{143},$$

however there are 11 solutions to the equation

$$11x = 22 \pmod{143}.$$

Luckily, it is very easy to test when such an equation has one, many or no solutions. We simply compute the greatest common divisor, or gcd, of a and N, i.e. $\gcd(a, N)$.

- If $\gcd(a, N) = 1$ then there is exactly one solution. We find the value c such that $a \cdot c = 1 \pmod{N}$ and then we compute $x = b \cdot c \pmod{N}$.
- If $g = \gcd(a, N) \neq 1$ and $\gcd(a, N)$ divides b then there are g solutions. Here we divide the whole equation by g to produce the equation
$$a' \cdot x' = b' \pmod{N'},$$
where $a' = a/g$, $b' = b/g$ and $N' = N/g$. If x' is a solution to the above equation then
$$x = x' + i \cdot N'$$
for $0 \leq i < g$ is a solution to the original one.
- Otherwise there are no solutions.

The case where $\gcd(a, N) = 1$ is so important we have a special name for it, we say a and N are relatively prime or coprime.

The number of integers in $\mathbb{Z}/N\mathbb{Z}$ which are relatively prime to N is given by the Euler ϕ function, $\phi(N)$. Given the prime factorization of N it is easy to compute the value of $\phi(N)$. If N has the prime factorization
$$N = \prod_{i=1}^{n} p_i^{e_i}$$
then
$$\phi(N) = \prod_{i=1}^{n} p_i^{e_i - 1}(p_i - 1).$$
Note, the last statement it is very important for cryptography: *Given the factorization of N it is easy to compute the value of $\phi(N)$*. The most important cases for the value of $\phi(N)$ in cryptography are:

(1) If p is prime then
$$\phi(p) = p - 1.$$
(2) If p and q are both prime and $p \neq q$ then
$$\phi(p \cdot q) = (p - 1)(q - 1).$$

1.3. Multiplicative Inverse Modulo N. We have just seen that when we wish to solve equations of the form
$$ax = b \pmod{N}$$
we reduce to the question of examining when an integer a modulo N has a multiplicative inverse, i.e. whether there is a number c such that
$$ac = ca = 1 \pmod{N}.$$
Such a value of c is often written a^{-1}. Clearly a^{-1} is the solution to the equation
$$ax = 1 \pmod{N}.$$
Hence, the inverse of a only exists when a and N are coprime, i.e. $\gcd(a, N) = 1$. Of particular interest is when N is a prime p, since then for all non-zero values of $a \in \mathbb{Z}/p\mathbb{Z}$ we always obtain a unique solution to
$$ax = 1 \pmod{p}.$$
Hence, if p is a prime then every non-zero element in $\mathbb{Z}/p\mathbb{Z}$ has a multiplicative inverse. A ring like $\mathbb{Z}/p\mathbb{Z}$ with this property is called a field.

DEFINITION 1.3 (Fields). *A field is a set with two operations $(G, \cdot, +)$ such that*

- *$(G, +)$ is an abelian group with identity denoted by 0,*
- *$(G \setminus \{0\}, \cdot)$ is an abelian group,*
- *$(G, \cdot, +)$ satisfies the distributive law.*

Hence, a field is a commutative ring for which every non-zero element has a multiplicative inverse. You have met fields before, for example consider the infinite fields of rational, real or complex numbers.

We define the set of all invertible elements in $\mathbb{Z}/N\mathbb{Z}$ by

$$(\mathbb{Z}/N\mathbb{Z})^* = \{x \in \mathbb{Z}/N\mathbb{Z} : \gcd(x, N) = 1\}$$

The * in A^* for any ring A refers to the largest subset of A which forms a group under multiplication. Hence, the set $(\mathbb{Z}/N\mathbb{Z})^*$ is a group with respect to multiplication and it has size $\phi(N)$.

In the special case when N is a prime p we have

$$(\mathbb{Z}/p\mathbb{Z})^* = \{1, \ldots, p-1\}$$

since every non-zero element of $\mathbb{Z}/p\mathbb{Z}$ is coprime to p. For an arbitrary field F the set F^* is equal to the set $F \setminus \{0\}$. To ease notation, for this very important case, define

$$\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z} = \{0, \ldots, p-1\}$$

and

$$\mathbb{F}_p^* = (\mathbb{Z}/p\mathbb{Z})^* = \{1, \ldots, p-1\}.$$

The set $\mathbb{F}_p$ is a finite field of characteristic p. In the next section we shall discuss a more general type of finite field, but for now recall the important point that the integers modulo N are only a field when N is a prime.

We end this section with the most important theorem in elementary group theory.

THEOREM 1.4 (Lagrange's Theorem). *If $(G, \cdot)$ is a group of order (size) $n = \#G$ then for all $a \in G$ we have $a^n = 1$.*

So if $x \in (\mathbb{Z}/N\mathbb{Z})^*$ then

$$x^{\phi(N)} = 1 \pmod{N}$$

since $\#(\mathbb{Z}/N\mathbb{Z})^* = \phi(N)$. This leads us to Fermat's Little Theorem, not to be confused with Fermat's Last Theorem which is something entirely different.

THEOREM 1.5 (Fermat's Little Theorem). *Suppose p is a prime and $a \in \mathbb{Z}$ then*

$$a^p = a \pmod{p}.$$

Fermat's Little Theorem is a special case of Lagrange's Theorem and will form the basis of one of the primality tests considered in a later chapter.

2. Finite Fields

The integers modulo a prime p are not the only types of finite field. In this section we shall introduce another type of finite field which is particularly important. At first reading you may wish to skip this section. We shall only be using these general forms of finite fields when discussing the Rijndael block cipher, stream ciphers based on linear feedback shift registers and when we look at elliptic curve based systems.

For this section we let p denote a prime number. Consider the set of polynomials in X whose coefficients are reduced modulo p. We denote this set $\mathbb{F}_p[X]$, which forms a ring with the natural definition of addition and multiplication.

Of particular interest is the case when $p = 2$, from which we draw all our examples in this section. For example, in $\mathbb{F}_2[X]$ we have

$$(1 + X + X^2) + (X + X^3) = 1 + X^2 + X^3,$$
$$(1 + X + X^2) \cdot (X + X^3) = X + X^2 + X^4 + X^5.$$

Just as with the integers modulo a number N, where the integers modulo N formed a ring, we can take a polynomial $f(X)$ and then the polynomials modulo $f(X)$ also form a ring. We denote this ring by

$$\mathbb{F}_p[X]/f(X)\mathbb{F}_p[X]$$

or more simply

$$\mathbb{F}_p[X]/(f(X)).$$

But to ease notation we will often write $\mathbb{F}_p[X]/f(X)$ for this latter ring. When $f(X) = X^4 + 1$ and $p = 2$ we have, for example,

$$(1 + X + X^2) \cdot (X + X^3) \pmod{X^4 + 1} = 1 + X^2$$

since

$$X + X^2 + X^4 + X^5 = (X + 1) \cdot (X^4 + 1) + (1 + X^2).$$

When checking the above equation you should remember we are working modulo two.

Recall, when we looked at the integers modulo N we looked at the equation

$$ax = b \pmod{N}.$$

We can consider a similar question for polynomials. Given a, b and f, all of which are polynomials in $\mathbb{F}_p[X]$, does there exist a solution α to the equation

$$a\alpha = b \pmod{f}?$$

With integers the answer depended on the greatest common divisor of a and f, and we counted three possible cases. A similar three cases can occur for polynomials, with the most important one being when a and f are coprime and so have greatest common divisor equal to one.

A polynomial is called irreducible if it has no proper factors other than itself and the constant polynomials. Hence, irreducibility of polynomials is the same as primality of numbers. Just as with the integers modulo N, when N was prime we obtained a finite field, so when $f(X)$ is irreducible the ring $\mathbb{F}_p[X]/f(X)$ also forms a finite field.

Consider the case $p = 2$ and the two different irreducible polynomials

$$f_1 = X^7 + X + 1$$

and

$$f_2 = X^7 + X^3 + 1.$$

Now, consider the two finite fields

$$F_1 = \mathbb{F}_2[X]/f_1(X) \text{ and } F_2 = \mathbb{F}_2[X]/f_2(X).$$

These both consist of the 2^7 binary polynomials of degree less than seven. Addition in these two fields is identical in that one just adds the coefficients of the polynomials modulo two. The only difference is in how multiplication is performed

$$(X^3 + 1) \cdot (X^4 + 1) \pmod{f_1(X)} = X^4 + X^3 + X,$$
$$(X^3 + 1) \cdot (X^4 + 1) \pmod{f_2(X)} = X^4.$$

A natural question arises as to whether these fields are 'really' different, or whether they just "look" different. In mathematical terms the question is whether the two fields are *isomorphic*. It turns out that they are isomorphic if there is a map

$$\phi : F_1 \longrightarrow F_2,$$

called a field isomorphism, which satisfies

$$\phi(\alpha+\beta)=\phi(\alpha)+\phi(\beta),$$
$$\phi(\alpha\cdot\beta)=\phi(\alpha)\cdot\phi(\beta).$$

Such an isomorphism exists for every two finite fields of the same order, although we will not show it here. To describe the map above you only need to show how to express a root of $f_2(X)$ in terms of a polynomial in the root of $f_1(X)$.

The above construction is in fact the only way of producing finite fields, hence all finite fields are essentially equal to polynomials modulo a prime and modulo an irreducible polynomial (for that prime). Hence, we have the following basic theorem

THEOREM 1.6. *There is (up to isomorphism) just one finite field of each prime power order.*

The notation we use for these fields is either $\mathbb{F}_q$ or $GF(q)$, with $q=p^d$ where d is the degree of the irreducible polynomial used to construct the finite field. We of course have $\mathbb{F}_p=\mathbb{F}_p[X]/X$. The notation $GF(q)$ means the Galois field of q elements. Finite fields are sometimes named after the 19th century French mathematician Galois. Galois had an interesting life, he accomplished most of his scientific work at an early age before dying in a dual.

There are a number of technical definitions associated with finite fields which we need to cover. Each finite field K contains a copy of the integers modulo p for some prime p, we call this prime the *characteristic* of the field, and often write this as char K. The subfield of integers modulo p of a finite field is called the prime subfield.

There is a map Φ called the p-th power *Frobenius map* defined for any finite field by

$$\Phi:\begin{cases}\mathbb{F}_q\longrightarrow\mathbb{F}_q\\ \alpha\longmapsto\alpha^p\end{cases}$$

where p is the characteristic of $\mathbb{F}_q$. The Frobenius map is an isomorphism of $\mathbb{F}_q$ with itself, such an isomorphism is called an automorphism. An interesting property is that the set of elements fixed by the Frobenius map is the prime field, i.e.

$$\{\alpha\in\mathbb{F}_q:\alpha^p=\alpha\}=\mathbb{F}_p.$$

Notice that this is a kind of generalization of Fermat's Little Theorem to finite fields. For any automorphism χ of a finite field the set of elements fixed by χ is a field, called the fixed field of χ. Hence the previous statement says that the fixed field of the Frobenius map is the prime field $\mathbb{F}_p$.

Not only does $\mathbb{F}_q$ contain a copy of $\mathbb{F}_p$ but $\mathbb{F}_{p^d}$ contains a copy of $\mathbb{F}_{p^e}$ for every value of e dividing d. In addition $\mathbb{F}_{p^e}$ is the fixed field of the automorphism Φ^e, i.e.

$$\{\alpha\in\mathbb{F}_{p^d}:\alpha^{p^e}=\alpha\}=\mathbb{F}_{p^e}.$$

Another interesting property is that if p is the characteristic of $\mathbb{F}_q$ then if we take any element $\alpha\in\mathbb{F}_q$ and add it to itself p times we obtain zero, e.g. in $\mathbb{F}_{49}$ we have

$$X+X+X+X+X+X+X=7X=0\pmod 7.$$

The non-zero elements of a finite field, usually denoted $\mathbb{F}_q^*$, form a cyclic finite abelian group. We call a generator of $\mathbb{F}_q^*$ a primitive element in the finite field. Such primitive elements always exist and so the multiplicative group is always cyclic. In other words there always exists an element $g\in\mathbb{F}_q$ such that every non-zero element α can be written as

$$\alpha=g^x$$

for some integer value of x.

As an example consider the field of eight elements defined by

$$\mathbb{F}_{2^3}=\mathbb{F}_2[X]/(X^3+X+1).$$

In this field there are seven non-zero elements namely

$$1, \alpha, \alpha+1, \alpha^2, \alpha^2+1, \alpha^2+\alpha, \alpha^2+\alpha+1$$

where α is a root of X^3+X+1. We see that α is a primitive element in $\mathbb{F}_{2^3}$ since

$$\begin{aligned}
\alpha^1 &= \alpha, \\
\alpha^2 &= \alpha^2, \\
\alpha^3 &= \alpha+1, \\
\alpha^4 &= \alpha^2+\alpha, \\
\alpha^5 &= \alpha^2+\alpha+1, \\
\alpha^6 &= \alpha^2+1, \\
\alpha^7 &= 1.
\end{aligned}$$

Notice that for a prime p this means that the integers modulo a prime also have a primitive element, since $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p$ is a finite field.

3. Basic Algorithms

There are several basic numerical algorithms or techniques which everyone should know since they occur in many places in this book. The ones we shall concentrate on here are

- Euclid's gcd algorithm,
- the Chinese Remainder Theorem,
- computing Jacobi and Legendre symbols.

3.1. Greatest Common Divisors. In the previous sections we said that when trying to solve

$$a \cdot x = b \pmod{N}$$

in integers, or

$$a\alpha = b \pmod{f}$$

for polynomials modulo a prime, we needed to compute the greatest common divisor. This was particularly important in determining whether $a \in \mathbb{Z}/N\mathbb{Z}$ or $a \in \mathbb{F}_p[X]/f$ had a multiplicative inverse or not, i.e. $\gcd(a, N) = 1$ or $\gcd(a, f) = 1$. We did not explain how this greatest common divisor is computed, neither did we explain how the inverse is to be computed when we know it exists. We shall now address this omission by explaining one of the oldest algorithms known to man, namely the Euclidean algorithm.

If we were able to factor a and N into primes, or a and f into irreducible polynomials, then computing the greatest common divisor would be particularly easy. For example if

$$\begin{aligned}
a &= 230\,895\,588\,646\,864 = 2^4 \cdot 157 \cdot 4513^3, \\
b &= 33\,107\,658\,350\,407\,876 = 2^2 \cdot 157 \cdot 2269^3 \cdot 4513,
\end{aligned}$$

then it is easy, from the factorization, to compute the gcd as

$$\gcd(a, b) = 2^2 \cdot 157 \cdot 4513 = 2\,834\,164.$$

However, factoring is an expensive operation for integers, but computing greatest common divisors is easy as we shall show. Although factoring for polynomials modulo a prime is very easy, it turns out that almost all algorithms to factor polynomials require access to an algorithm to compute greatest common divisors. Hence, in both situations we need to be able to compute greatest common divisors without recourse to factoring.

3.1.1. *Euclidean Algorithm:* In the following we will consider the case of integers only, the generalization to polynomials is easy since both integers and polynomials allow Euclidean division. For integers Euclidean division is the operation of, given a and b, finding q and r with $0 \leq r < |b|$ such that

$$a = q \cdot b + r.$$

For polynomials Euclidean division is given polynomials f, g finding polynomials q, r with $0 \leq \deg r < \deg g$ such that

$$f = q \cdot g + r.$$

To compute the gcd of $r_0 = a$ and $r_1 = b$ we compute $r_2, r_3, r_4, \ldots$ as follows;

$$\begin{aligned} r_2 &= q_1 r_1 - r_0 \\ r_3 &= q_2 r_2 - r_1 \\ &\vdots \quad \vdots \\ r_m &= q_{m-1} r_{m-1} - r_{m-2} \\ r_{m+1} &= q_m r_m. \end{aligned}$$

If d divides a and b then d divides r_2, r_3, r_4 and so on. Hence

$$\gcd(a, b) = \gcd(r_0, r_1) = \gcd(r_1, r_2) = \cdots = \gcd(r_{m-1}, r_m) = r_m.$$

As an example of this algorithm we want to show that

$$3 = \gcd(21, 12).$$

Using the Euclidean algorithm we compute $\gcd(21, 12)$ in the steps

$$\begin{aligned} \gcd(21, 12) &= \gcd(21 \pmod{12}, 12) \\ &= \gcd(9, 12) \\ &= \gcd(12 \pmod{9}, 9) \\ &= \gcd(3, 9) \\ &= \gcd(9 \pmod{3}, 3) \\ &= \gcd(0, 3) = 3. \end{aligned}$$

Or, as an example with larger numbers,

$$\begin{aligned} \gcd(1\,426\,668\,559\,730,\ 810\,653\,094\,756) &= \gcd(810\,653\,094\,756,\ 616\,015\,464\,974), \\ &= \gcd(616\,015\,464\,974,\ 194\,637\,629\,782), \\ &= \gcd(194\,637\,629\,782,\ 32\,102\,575\,628), \\ &= \gcd(32\,102\,575\,628,\ 2\,022\,176\,014), \\ &= \gcd(2\,022\,176\,014,\ 1\,769\,935\,418), \\ &= \gcd(1\,769\,935\,418,\ 252\,240\,596), \\ &= \gcd(252\,240\,596,\ 4\,251\,246), \\ &= \gcd(4\,251\,246,\ 1\,417\,082), \\ &= \gcd(1\,417\,082,\ 0), \\ &= 1\,417\,082. \end{aligned}$$

The Euclidean algorithm essentially works because the map

$$(a, b) \longmapsto (a \pmod{b}, b),$$

for $a \geq b$ is a gcd preserving mapping. The trouble is that computers find it much easier to add and multiply numbers than to take remainders or quotients. Hence, implementing a gcd algorithm

with the above gcd preserving mapping will usually be very inefficient. Fortunately, there are a number of other gcd preserving mappings, for example

$$(a,b) \longmapsto \begin{cases} ((a-b)/2, b) & \text{If } a \text{ and } b \text{ are odd.} \\ (a/2, b) & \text{If } a \text{ is even and } b \text{ is odd.} \\ (a, b/2) & \text{If } a \text{ is odd and } b \text{ is even.} \end{cases}$$

Recall that computers find it easy to divide by two, since in binary this is accomplished by a cheap bit shift operation. This latter mapping gives rise to the binary Euclidean algorithm, which is the one usually implemented on a computer. Essentially, this algorithm uses the above gcd preserving mapping after first removing any power of two in the gcd. Algorithm 1.1 explains how this works, on input of two positive integers a and b.

Algorithm 1.1: Binary Euclidean Algorithm

```
g = 1
/* Remove powers of two from the gcd */
while (a mod 2 = 0) and (b mod 2 = 0) do
    a = a/2
    b = b/2
    g = 2 · g
end
/* At least one of a and b is now odd */
while a ≠ 0 do
    while a mod 2 = 0 do a = a/2
    while b mod 2 = 0 do b = b/2
    /* Now both a and b are odd */
    if a ≥ b then a = (a − b)/2
    else b = (b − a)/2
end
return g · b
```

3.1.2. *Extended Euclidean Algorithm:* Using the Euclidean algorithm we can determine when a has an inverse modulo m by testing whether

$$\gcd(a, m) = 1.$$

But we still do not know how to determine the inverse when it exists. To do this we use a variant of Euclid's gcd algorithm, called the extended Euclidean algorithm. Recall we had

$$r_{i-2} = q_{i-1} r_{i-1} + r_i$$

with $r_m = \gcd(r_0, r_1)$. Now we unwind the above and write each r_i, for $i \geq 2$, in terms of a and b. For example

$$\begin{aligned}
r_2 &= r_0 - q_1 r_1 = a - q_1 b \\
r_3 &= r_1 - q_2 r_2 = b - q_2(a - q_1 b) = -q_2 a + (1 + q_1 q_2) b \\
&\vdots \quad \vdots \\
r_{i-2} &= s_{i-2} a + t_{i-2} b \\
r_{i-1} &= s_{i-1} a + t_{i-1} b \\
r_i &= r_{i-2} - q_{i-1} r_{i-1} \\
&= a(s_{i-2} - q_{i-1} s_{i-1}) + b(t_{i-2} - q_{i-1} t_{i-1}) \\
&\vdots \quad \vdots \\
r_m &= s_m a + t_m b.
\end{aligned}$$

The extended Euclidean algorithm takes as input a and b and outputs r_m, s_m and t_m such that

$$r_m = \gcd(a, b) = s_m a + t_m b.$$

Hence, we can now solve our original problem of determining the inverse of a modulo N, when such an inverse exists. We first apply the extended Euclidean algorithm to a and N so as to compute d, x, y such that

$$d = \gcd(a, N) = xa + yN.$$

We can solve the equation $ax = 1 \pmod N$, since we have $d = xa + yN = xa \pmod N$. Hence, we have a solution $x = a^{-1}$, precisely when $d = 1$.

As an example suppose we wish to compute the inverse of 7 modulo 19. We first set $r_0 = 7$ and $r_1 = 19$ and then we compute

$$\begin{aligned}
r_2 &= 5 = 19 - 2 \cdot 7 \\
r_3 &= 2 = 7 - 5 = 7 - (19 - 2 \cdot 7) = -19 + 3 \cdot 7 \\
r_4 &= 1 = 5 - 2 \cdot 2 = (19 - 2 \cdot 7) - 2 \cdot (-19 + 3 \cdot 7) = 3 \cdot 19 - 8 \cdot 7.
\end{aligned}$$

Hence,

$$1 = -8 \cdot 7 \pmod{19}$$

and so

$$7^{-1} = -8 = 11 \pmod{19}.$$

3.2. Chinese Remainder Theorem (CRT). The Chinese Remainder Theorem, or CRT, is also a very old piece of mathematics, which dates back at least 2000 years. We shall use the CRT in a few places, for example to improve the performance of the decryption operation of RSA and in a number of other protocols. In a nutshell the CRT states that if we have the two equations

$$x = a \pmod N \text{ and } x = b \pmod M$$

then there is a unique solution modulo $M \cdot N$ if and only if $\gcd(N, M) = 1$. In addition it gives a method to easily find the solution. For example if the two equations are given by

$$\begin{aligned}
x &= 4 \pmod 7, \\
x &= 3 \pmod 5,
\end{aligned}$$

then we have

$$x = 18 \pmod{35}.$$

It is easy to check that this is a solution, since $18 \pmod 7 = 4$ and $18 \pmod 5 = 3$. But how did we produce this solution?

We shall first show how this can be done naively from first principles and then we shall give the general method. We have the equations

$$x = 4 \pmod 7 \text{ and } x = 3 \pmod 5.$$

Hence for some u we have

$$x = 4 + 7u \text{ and } x = 3 \pmod 5.$$

Putting these latter two equations into one gives,

$$4 + 7u = 3 \pmod 5.$$

We then rearrange the equation to find

$$2u = 7u = 3 - 4 = 4 \pmod 5.$$

Now since $\gcd(2, 5) = \gcd(7, 5) = 1$ we can solve the above equation for u. First we compute $2^{-1} \pmod 5 = 3$, since $2 \cdot 3 = 6 = 1 \pmod 5$. Then we compute the value of $u = 3 \cdot 4 \pmod 5$. Then substituting this value of u back into our equation for x gives the solution

$$x = 4 + 7u = 4 + 7 \cdot 2 = 18.$$

The case of two equations is so important we now give a general formula. We assume that $\gcd(N, M) = 1$, and that we are given the equations

$$x = a \pmod M \text{ and } x = b \pmod N.$$

We first compute

$$T = M^{-1} \pmod N$$

which is possible since we have assumed $\gcd(N, M) = 1$. We then compute

$$u = (b - a)T \pmod N.$$

The solution modulo $M \cdot N$ is then given by

$$x = a + uM.$$

To see this always works we compute

$$\begin{aligned} x \pmod M &= a + uM \pmod M \\ &= a, \\ x \pmod N &= a + uM \pmod N \\ &= a + (b - a)TM \pmod N \\ &= a + (b - a)M^{-1}M \pmod N \\ &= a + (b - a) \pmod N \\ &= b. \end{aligned}$$

Now we turn to the general case of the CRT where we consider more than two equations at once. Let $m_1, \ldots, m_r$ be pairwise relatively prime and let $a_1, \ldots, a_r$ be given. We want to find x modulo $M = m_1 m_2 \cdots m_r$ such that

$$x = a_i \pmod{m_i} \text{ for all } i.$$

The Chinese Remainder Theorem guarantees a unique solution given by

$$x = \sum_{i=1}^{r} a_i M_i y_i \pmod M$$

where

$$M_i = M/m_i,$$
$$y_i = M_i^{-1} \pmod{m_i}.$$

As an example suppose we wish to find the unique x modulo

$$M = 1001 = 7 \cdot 11 \cdot 13$$

such that

$$x = 5 \pmod 7,$$
$$x = 3 \pmod{11},$$
$$x = 10 \pmod{13}.$$

We compute

$$M_1 = 143, \quad y_1 = 5,$$
$$M_2 = 91, \quad y_2 = 4,$$
$$M_3 = 77, \quad y_3 = 12.$$

Then, the solution is given by

$$\begin{aligned} x &= \sum_{i=1}^{r} a_i M_i y_i \pmod M \\ &= 715 \cdot 5 + 364 \cdot 3 + 924 \cdot 10 \pmod{1001} \\ &= 894. \end{aligned}$$

3.3. Legendre and Jacobi Symbols. Let p denote a prime, greater than two. Consider the mapping

$$\mathbb{F}_p \longrightarrow \mathbb{F}_p$$
$$\alpha \longmapsto \alpha^2.$$

This mapping is exactly two-to-one on the non-zero elements of $\mathbb{F}_p$. So if an element x in $\mathbb{F}_p$ has a square root, then it has exactly two square roots (unless $x = 0$) and exactly half of the elements of $\mathbb{F}_p^*$ are squares. The set of squares in $\mathbb{F}_p^*$ are called the *quadratic residues* and they form a subgroup, of order $(p-1)/2$ of the multiplicative group $\mathbb{F}_p^*$. The elements of $\mathbb{F}_p^*$ which are not squares are called the *quadratic non-residues*.

To make it easy to detect squares modulo p we define the *Legendre symbol*

$$\left(\frac{a}{p}\right).$$

This is defined to be equal to 0 if p divides a, it is equal to $+1$ if a is a quadratic residue and it is equal to -1 if a is a quadratic non-residue.

It is easy to compute the Legendre symbol, for example via

$$\left(\frac{a}{p}\right) = a^{(p-1)/2} \pmod p.$$

However, using the above formula turns out to be very inefficient. In practice one uses the *law of quadratic reciprocity*

$$\left(\frac{q}{p}\right) = \left(\frac{p}{q}\right)(-1)^{(p-1)(q-1)/4}. \tag{1}$$

In other words we have

$$\left(\frac{q}{p}\right) = \begin{cases} -\left(\frac{p}{q}\right) & \text{If } p = q = 3 \pmod 4, \\ \left(\frac{p}{q}\right) & \text{Otherwise} \end{cases}$$

Using this law with the following additional formulae gives rise to a recursive algorithm

$$\left(\frac{q}{p}\right) = \left(\frac{q \pmod p}{p}\right), \tag{2}$$

$$\left(\frac{q \cdot r}{p}\right) = \left(\frac{q}{p}\right) \cdot \left(\frac{r}{p}\right), \tag{3}$$

$$\left(\frac{2}{p}\right) = (-1)^{(p^2-1)/8}. \tag{4}$$

Assuming we can factor, we can now compute the Legendre symbol

$$\begin{aligned} \left(\frac{15}{17}\right) &= \left(\frac{3}{17}\right) \cdot \left(\frac{5}{17}\right) \text{ by Equation (3)} \\ &= \left(\frac{17}{3}\right) \cdot \left(\frac{17}{5}\right) \text{ by Equation (1)} \\ &= \left(\frac{2}{3}\right) \cdot \left(\frac{2}{5}\right) \text{ by Equation (2)} \\ &= (-1) \cdot (-1)^3 \text{ by Equation (4)} \\ &= 1. \end{aligned}$$

In a moment we shall see a more efficient algorithm which does not require us to factor integers.

Computing square roots of elements in $\mathbb{F}_p^*$, when the square root exists turns out to be an easy task. Algorithm 1.2 gives one method, called Shanks' Algorithm, of computing the square root of a modulo p, when such a square root exists.

When $p = 3 \pmod 4$, instead of the above algorithm, we can use the following formulae

$$x = a^{(p+1)/4} \pmod p,$$

which has the advantage of being deterministic and more efficient than the general method of Shanks. That this formula works is because

$$x^2 = a^{(p+1)/2} = a^{(p-1)/2} \cdot a = \left(\frac{a}{p}\right) \cdot a = a$$

where the last equality holds since we have assumed that a is a quadratic residue modulo p and so it has Legendre symbol equal to one.

The Legendre symbol above is only defined when its denominator is a prime, but there is a generalization to composite denominators called the *Jacobi symbol*. Suppose $n \geq 3$ is odd and

$$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$$

then the Jacobi symbol

$$\left(\frac{a}{n}\right)$$

is defined in terms of the Legendre symbol by

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{e_1} \left(\frac{a}{p_2}\right)^{e_2} \cdots \left(\frac{a}{p_k}\right)^{e_k}.$$

Algorithm 1.2: Shanks' algorithm for square roots modulo p

Choose a random n until one is found such that
$$\left(\frac{n}{p}\right) = -1$$
Let e, q be integers such that q is odd and $p - 1 = 2^e q$
$y = n^q \pmod{p}$
$r = e$
$x = a^{(q-1)/2} \pmod{p}$
$b = ax^2 \pmod{p}$
$x = ax \pmod{p}$
while $b \neq 1 \pmod{p}$ **do**
  Find the smallest m such that $b^{2^m} = 1 \pmod{p}$
  $t = y^{2^{r-m-1}} \pmod{p}$
  $y = t^2 \pmod{p}$
  $r = m$
  $x = xt \pmod{p}$
  $b = by \pmod{p}$
end
return x

The Jacobi symbol can be computed using a similar method to the Legendre symbol by making use of the identity, derived from the law of quadratic reciprocity,
$$\left(\frac{a}{n}\right) = \left(\frac{2}{n}\right)^e \left(\frac{n \pmod{a_1}}{a_1}\right) (-1)^{(a_1-1)(n-1)/4}.$$
where $a = 2^e a_1$ and a_1 is odd. We also require the identities, for n odd,
$$\left(\frac{1}{n}\right) = 1,$$
$$\left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8},$$
$$\left(\frac{-1}{n}\right) = (-1)^{(n-1)/2}.$$
This now gives us a fast algorithm, which does not require factoring of integers, to determine the Jacobi symbol, and so the Legendre symbol in the case where the denominator is prime. The only factoring required is that of extracting the even part of a number:
$$\begin{aligned}\left(\frac{15}{17}\right) &= (-1)^{56}\left(\frac{17}{15}\right)\\ &= \left(\frac{2}{15}\right)\\ &= (-1)^{28} = 1.\end{aligned}$$

Recall the Legendre symbol $\left(\frac{a}{p}\right)$ tells us whether a is a square modulo p, for p a prime. Alas, the Jacobi symbol $\left(\frac{a}{n}\right)$ does not tell us the whole story about whether a is a square modulo n, when n is a composite. If a is a square modulo n then the Jacobi symbol will be equal to plus one, however if the Jacobi symbol is equal to plus one then it is not always true that a is a square.

Let $n \geq 3$ be odd and let the set of squares in $(\mathbb{Z}/n\mathbb{Z})^*$ be denoted

$$Q_n = \{x^2 \pmod{n} : x \in (\mathbb{Z}/n\mathbb{Z})^*\}.$$

Now let J_n denote the set of elements with Jacobi symbol equal to plus one, i.e.

$$J_n = \left\{x \in (\mathbb{Z}/n\mathbb{Z})^* : \left(\frac{a}{n}\right) = 1\right\}.$$

The set of pseudo-squares is the difference $J_n \setminus Q_n$.

There are two important cases for cryptography, either n is prime or n is the product of two primes:

- n is a prime p.
  - $Q_n = J_n$.
  - $\#Q_n = (n-1)/2$.
- n is the product of two primes, $n = p \cdot q$.
  - $Q_n \subset J_n$.
  - $\#Q_n = \#(J_n \setminus Q_n) = (p-1)(q-1)/4$.

The sets Q_n and J_n will be seen to be important in a number of algorithms and protocols, especially in the case where n is a product of two primes.

Finally, we look at how to compute a square root modulo a composite number $n = p \cdot q$. Suppose we wish to compute the square root of a modulo n. We assume we know p and q, and that a really is a square modulo n, which can be checked by demonstrating that

$$\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = 1.$$

We first compute the square root of a modulo p, call this s_p. Then we compute the square root of a modulo q, call this s_q. Finally to deduce the square root modulo n, we apply the Chinese Remainder Theorem to the equations

$$x = s_p \pmod{p} \text{ and } x = s_q \pmod{q}.$$

As an example suppose we wish to compute the square root of $a = 217$ modulo $n = 221 = 13 \cdot 17$. Now the square root of a modulo 13 and 17 is given by

$$s_{13} = 3 \text{ and } s_{17} = 8.$$

Applying the Chinese Remainder Theorem we find

$$s = 42$$

and we can check that s really is a square root by computing

$$s^2 = 42^2 = 217 \pmod{n}.$$

There are three other square roots, since n has two prime factors. These other square roots are obtained by applying the Chinese Remainder Theorem to the three other equations

$$\begin{aligned} s_{13} &= 10, \quad s_{17} = 8, \\ s_{13} &= 3, \quad s_{17} = 9, \\ s_{13} &= 10, \quad s_{17} = 9, \end{aligned}$$

Hence, all four square roots of 217 modulo 221 are given by

$$42,\ 94,\ 127 \text{ and } 179.$$

4. Probability

At some points we will need a basic understanding of elementary probability theory. In this section we summarize the theory we require and give a few examples. Most readers should find this a revision of the type of probability encountered in high school.

A *random variable* is a variable X which takes certain values with given probabilities. If X takes on the value s with probability 0.01 we write this as

$$p(X = s) = 0.01.$$

As an example, let T be the random variable representing tosses of a fair coin, we then have the probabilities

$$\begin{aligned} p(T = \text{Heads}) &= \frac{1}{2}, \\ p(T = \text{Tails}) &= \frac{1}{2}. \end{aligned}$$

As another example let E be the random variable representing letters in English text. An analysis of a large amount of English text allows us to approximate the relevant probabilities by

$$\begin{aligned} p(E = a) &= 0.082, \\ &\vdots \\ p(E = e) &= 0.127, \\ &\vdots \\ p(E = z) &= 0.001. \end{aligned}$$

Basically if X is a discrete random variable and $p(X = x)$ is the *probability distribution* then we have the two following properties:

$$\begin{aligned} p(X = x) &\geq 0, \\ \sum_x p(X = x) &= 1. \end{aligned}$$

It is common to illustrate examples from probability theory using a standard deck of cards. We shall do likewise and let V denote the random variable that a card is a particular value, let S denote the random variable that a card is a particular suit and let C denote the random variable of the colour of a card. So for example

$$\begin{aligned} p(C = \text{Red}) &= \frac{1}{2}, \\ p(V = \text{Ace of Clubs}) &= \frac{1}{52}, \\ p(S = \text{Clubs}) &= \frac{1}{4}. \end{aligned}$$

Let X and Y be two random variables, where $p(X = x)$ is the probability that X takes the value x and $p(Y = y)$ is the probability that Y takes the value y. The *joint probability* $p(X = x, Y = y)$ is defined as the probability that X takes the value x and Y takes the value y. So if we let $X = C$

and $Y = S$ then we have

$$\begin{aligned} &p(C = \text{Red}, S = \text{Club}) = 0, && p(C = \text{Red}, S = \text{Diamonds}) = \frac{1}{4}, \\ &p(C = \text{Red}, S = \text{Hearts}) = \frac{1}{4}, && p(C = \text{Red}, S = \text{Spades}) = 0, \\ &p(C = \text{Black}, S = \text{Club}) = \frac{1}{4}, && p(C = \text{Black}, S = \text{Diamonds}) = 0, \\ &p(C = \text{Black}, S = \text{Hearts}) = 0, && p(C = \text{Black}, S = \text{Spades}) = \frac{1}{4}. \end{aligned}$$

Two random variables X and Y are said to be *independent* if, for all values of x and y,

$$p(X = x, Y = y) = p(X = x) \cdot p(Y = y).$$

Hence, the random variables C and S are not independent. As an example of independent random variables consider the two random variables, T_1 the value of the first toss of an unbiased coin and T_2 the value of a second toss of the coin. Since, assuming standard physical laws, the toss of the first coin does not affect the outcome of the toss of the second coin, we say that T_1 and T_2 are independent. This is confirmed by the joint probability distribution

$$\begin{aligned} &p(T_1 = H, T_2 = H) = \frac{1}{4}, && p(T_1 = H, T_2 = T) = \frac{1}{4}, \\ &p(T_1 = T, T_2 = H) = \frac{1}{4}, && p(T_1 = T, T_2 = T) = \frac{1}{4}. \end{aligned}$$

4.1. Bayes' Theorem. The *conditional probability* $p(X = x|Y = y)$ of two random variables X and Y is defined as the probability that X takes the value x given that Y takes the value y.

Returning to our random variables based on a pack of cards we have

$$p(S = \text{Spades}|C = \text{Red}) = 0$$

and

$$p(V = \text{Ace of Spades}|C = \text{Black}) = \frac{1}{26}.$$

The first follows since if we know a card is red, then the probability that it is a spade is zero, since a red card cannot be a spade. The second follows since if we know a card is black then we have restricted the set of cards in half, one of which is the ace of spades.

The following is one of the most crucial statements in probability theory

THEOREM 1.7 (Bayes' Theorem). *If* $p(Y = y) > 0$ *then*

$$\begin{aligned} p(X = x|Y = y) &= \frac{p(X = x) \cdot p(Y = y|X = x)}{p(Y = y)} \\ &= \frac{p(X = x, Y = y)}{p(Y = y)}. \end{aligned}$$

We can apply Bayes' Theorem to our examples above as follows

$$\begin{aligned} p(S = \text{Spades}|C = \text{Red}) &= \frac{p(S = \text{Spades}, C = \text{Red})}{p(C = \text{Red})} \\ &= 0 \cdot \left(\frac{1}{4}\right)^{-1} = 0. \end{aligned}$$

$$\begin{aligned} p(V = \text{Ace of Spades}|C = \text{Black}) &= \frac{p(V = \text{Ace of Spades}, C = \text{Black})}{p(C = \text{Black})} \\ &= \frac{1}{52} \cdot \left(\frac{1}{2}\right)^{-1} \\ &= \frac{2}{52} = \frac{1}{26}. \end{aligned}$$

If X and Y are independent then we have

$$p(X = x|Y = y) = p(X = x),$$

i.e. the value which X takes does not depend on the value that Y takes.

4.2. Birthday Paradox. Another useful result from elementary probability theory that we require is the *birthday paradox*. Suppose a bag has m balls in it, all of different colours. We draw one ball at a time from the bag and write down its colour, we then replace the ball in the bag and draw again.

If we define

$$m^{(n)} = m \cdot (m-1) \cdot (m-2) \cdots (m-n+1)$$

then the probability, after n balls have been taken out of the bag, that we have obtained at least one matching colour (or coincidence) is

$$1 - \frac{m^{(n)}}{m^n}.$$

As m becomes larger the expected number of balls we have to draw before we obtain the first coincidence is

$$\sqrt{\frac{\pi m}{2}}.$$

To see why this is called the birthday paradox consider the probability of two people in a room sharing the same birthday. Most people initially think that this probability should be quite low, since they are thinking of the probability that someone in the room shares the same birthday as them. One can now easily compute that the probability of at least two people in a room of 23 people having the same birthday is

$$1 - \frac{365^{(23)}}{365^{23}} \approx 0.507.$$

In fact this probability increases quite quickly since in a room of 30 people we obtain a probability of approximately 0.706, and in a room of 100 people we obtain a probability of over 0.999 999 6.

Chapter Summary

- A group is a set with an operation which has an identity, is associative and every element has an inverse. Modular arithmetic, both addition and multiplication, provides examples of groups. However, for multiplication we need to be careful which set of numbers we take when defining a group with respect to modular multiplication.
- A ring is a set with two operations which behaves like the set of integers under addition and multiplication. Modular arithmetic is an example of a ring.
- A field is a ring in which all non-zero elements have a multiplicative inverse. Integers modulo a prime are examples of fields.

- Multiplicative inverses for modular arithmetic can be found using the extended Euclidean algorithm.
- Sets of simultaneous linear modular equations can be solved using the Chinese Remainder Theorem.
- Square elements modulo a prime can be detected using the Legendre symbol, square roots can be efficiently computed using Shanks' Algorithm.
- Square elements and square roots modulo a composite can be determined efficiently as long as one knows the factorization of the modulus.
- Bayes' theorem allows us to compute conditional probabilities.
- The birthday paradox allows us to estimate how quickly collisions occur when one repeatedly samples from a finite space.

Further Reading

Bach and Shallit is the best introductory book I know which deals with Euclid's algorithm and finite fields. It contains a lot of historical information, plus excellent pointers to the relevant research literature. Whilst aimed in some respects at Computer Scientists, Bach and Shallit's book may be a little too mathematical for some. For a more traditional introduction to the basic discrete mathematics we shall need, at the level of a first year course in Computer Science, see the books by Biggs or Rosen.

E. Bach and J. Shallit. *Algorithmic Number Theory. Volume 1: Efficient Algorithms.* MIT Press, 1996.

N.L. Biggs. *Discrete Mathematics.* Oxford University Press, 1989.

K.H. Rosen. *Discrete Mathematics and its Applications.* McGraw-Hill, 1999.

CHAPTER 2

Elliptic Curves

Chapter Goals

- To describe what an elliptic curve is.
- To explain the basic mathematics behind elliptic curve cryptography.
- To show how projective coordinates can be used to improve computational efficiency.
- To show how point compression can be used to improve communications efficiency.

1. Introduction

This chapter is devoted to introducing elliptic curves. Some of the more modern public key systems make use of elliptic curves since they can offer improved efficiency and bandwidth. Since much of this book can be read by understanding that an elliptic curve provides another finite abelian group in which one can pose a discrete logarithm problem, you may decide to skip this chapter on an initial reading.

Let K be any field. The projective plane $\mathbb{P}^2(K)$ over K is defined as the set of triples

$$(X, Y, Z)$$

where $X, Y, Z \in K$ are not all simultaneously zero. On these triples is defined an equivalence relation

$$(X, Y, Z) \equiv (X', Y', Z')$$

if there exists a $\lambda \in K$ such that

$$X = \lambda X',\ Y = \lambda Y' \text{ and } Z = \lambda Z'.$$

So, for example, if $K = \mathbb{F}_7$, the finite field of seven elements, then the two points

$$(4, 1, 1) \text{ and } (5, 3, 3)$$

are equivalent. Such a triple is called a projective point.

An *elliptic curve* over K will be defined as the set of solutions in the projective plane $\mathbb{P}^2(K)$ of a homogeneous Weierstrass equation of the form

$$E : Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3,$$

with $a_1, a_2, a_3, a_4, a_6 \in K$. This equation is also referred to as the long Weierstrass form. Such a curve should be non-singular in the sense that, if the equation is written in the form $F(X, Y, Z) = 0$, then the partial derivatives of the curve equation

$$\partial F/\partial X,\ \partial F/\partial Y \text{ and } \partial F/\partial Z$$

should not vanish simultaneously at any point on the curve.

The set of K-rational points on E, i.e. the solutions in $\mathbb{P}^2(K)$ to the above equation, is denoted by $E(K)$. Notice, that the curve has exactly one rational point with coordinate Z equal to zero, namely $(0, 1, 0)$. This is the point at infinity, which will be denoted by $\mathcal{O}$.

For convenience, we will most often use the affine version of the Weierstrass equation, given by

$$E : Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6, \tag{5}$$

where $a_i \in K$.

The K-rational points in the affine case are the solutions to E in K^2, plus the point at infinity $\mathcal{O}$. Although most protocols for elliptic curve based cryptography make use of the affine form of a curve, it is often computationally important to be able to switch to projective coordinates. Luckily this switch is easy:

- The point at infinity always maps to the point at infinity in either direction.
- To map a projective point (X, Y, Z) which is not at infinity, so $Z \neq 0$, to an affine point we simply compute $(X/Z, Y/Z)$.
- To map an affine point (X, Y), which is not at infinity, to a projective point we take a random non-zero $Z \in K$ and compute $(X \cdot Z, Y \cdot Z, Z)$.

As we shall see later it is often more convenient to use a slightly modified form of projective point where the projective point (X, Y, Z) represents the affine point $(X/Z^2, Y/Z^3)$.

Given an elliptic curve defined by Equation (5), it is useful to define the following constants for use in later formulae:

$$\begin{aligned}
b_2 &= a_1^2 + 4a_2, \\
b_4 &= a_1a_3 + 2a_4, \\
b_6 &= a_3^2 + 4a_6, \\
b_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2, \\
c_4 &= b_2^2 - 24b_4, \\
c_6 &= -b_2^3 + 36b_2b_4 - 216b_6.
\end{aligned}$$

The discriminant of the curve is defined as

$$\Delta = -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6.$$

When char $K \neq 2, 3$ the discriminant can also be expressed as

$$\Delta = (c_4^3 - c_6^2)/1728.$$

Notice that $1728 = 2^63^3$ so, if the characteristic is not equal to 2 or 3, dividing by this latter quantity makes sense. A curve is then non-singular if and only if $\Delta \neq 0$; from now on we shall assume that $\Delta \neq 0$ in all our discussions.

When $\Delta \neq 0$, the j-invariant of the curve is defined as

$$j(E) = c_4^3/\Delta.$$

As an example, which we shall use throughout this chapter, we consider the elliptic curve

$$E : Y^2 = X^3 + X + 3$$

defined over the field $\mathbb{F}_7$. Computing the various quantities above we find that we have

$$\Delta = 3 \text{ and } j(E) = 5.$$

The j-invariant is closely related to the notion of elliptic curve isomorphism. Two elliptic curves defined by Weierstrass equations E (with variables X, Y) and E' (with variables X', Y') are isomorphic over K if and only if there exist constants $r, s, t \in K$ and $u \in K^*$, such that the change of variables

$$X = u^2X' + r\ , \quad Y = u^3Y' + su^2X' + t$$

transforms E into E'. Such an isomorphism defines a bijection between the set of rational points in E and the set of rational points in E'. Notice that isomorphism is defined relative to the field K.

As an example consider again the elliptic curve

$$E : Y^2 = X^3 + X + 3$$

over the field $\mathbb{F}_7$. Now make the change of variables defined by $[u, r, s, t] = [2, 3, 4, 5]$, i.e.

$$X = 4X' + 3 \text{ and } Y = Y' + 2X' + 5.$$

We then obtain the isomorphic curve

$$E' : Y'^2 + 4X'Y' + 3Y' = X'^3 + X' + 1,$$

and we have

$$j(E) = j(E') = 5.$$

Curve isomorphism is an equivalence relation. The following lemma establishes the fact that, over the algebraic closure $\overline{K}$, the j-invariant characterizes the equivalence classes in this relation.

LEMMA 2.1. *Two elliptic curves that are isomorphic over K have the same j-invariant. Conversely, two curves with the same j-invariant are isomorphic over $\overline{K}$.*

But curves with the same j-invariant may not necessarily be isomorphic over the ground field. For example, consider the elliptic curve, also over $\mathbb{F}_7$,

$$E'' : Y''^2 = X''^3 + 4X'' + 4.$$

This has j-invariant equal to 5 so it is isomorphic to E, but it is not isomorphic over $\mathbb{F}_7$ since the change of variable required is given by

$$X = 3X'' \text{ and } Y = \sqrt{6}Y''.$$

However, $\sqrt{6} \notin \mathbb{F}_7$. Hence, we say both E and E'' are defined over $\mathbb{F}_7$, but they are isomorphic over $\mathbb{F}_{7^2} = \mathbb{F}_7[\sqrt{6}]$.

2. The Group Law

Assume, for the moment, that char $K \neq 2, 3$, and consider the change of variables given by

$$X = X' - \frac{b_2}{12},$$
$$Y = Y' - \frac{a_1}{2}\left(X' - \frac{b_2}{12}\right) - \frac{a_3}{2}.$$

This change of variables transforms the long Weierstrass form given in Equation (5) to the equation of an isomorphic curve given in short Weierstrass form,

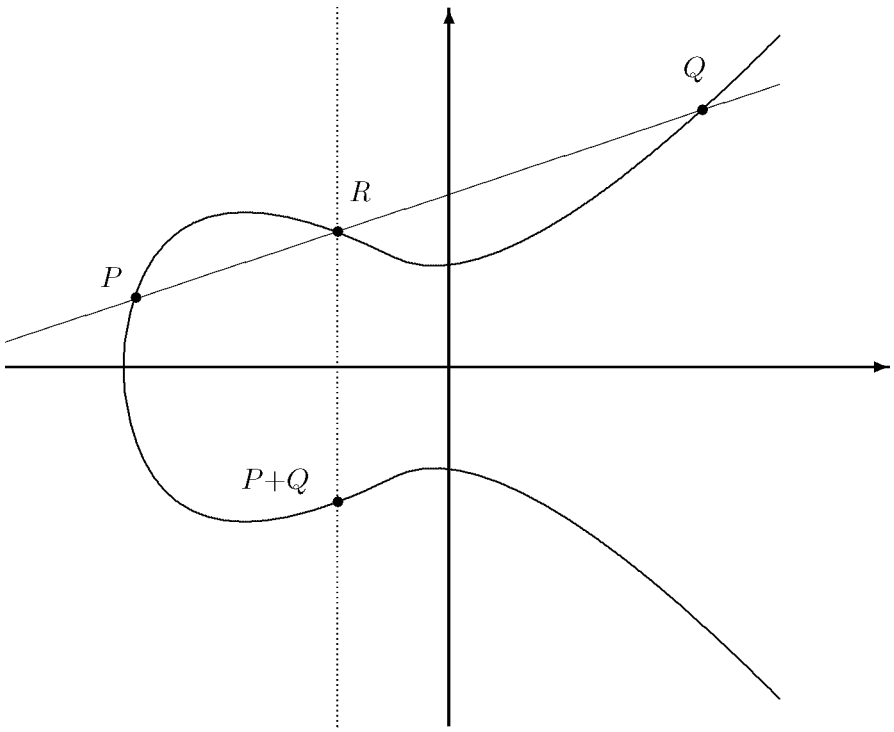
$$E : Y^2 = X^3 + aX + b,$$

for some $a, b \in K$. One can then define a group law on an elliptic curve using the chord-tangent process.

The chord process is defined as follows, see Fig. 1 for a diagrammatic description. Let P and Q be two distinct points on E. The straight line joining P and Q must intersect the curve at one further point, say R, since we are intersecting a line with a cubic curve. The point R will also be defined over the same field of definition as the curve and the two points P and Q. If we then reflect R in the x-axis we obtain another point over the same field which we shall call $P + Q$.

The tangent process is given diagrammatically in Fig. 2 or as follows. Let P denote a point on the curve E. We take the tangent to the curve at P. Such a line must intersect E in at most one other point, say R, as the elliptic curve E is defined by a cubic equation. Again we reflect R in

FIGURE 1. Adding two points on an elliptic curve



the x-axis to obtain a point which we call $[2]P = P + P$. If the tangent to the point is vertical, it 'intersects' the curve at the point at infinity and $P + P = \mathcal{O}$, and P is said to be a point of order 2.

One can show that the chord-tangent process turns E into an abelian group with the point at infinity $\mathcal{O}$ being the zero. The above definition can be easily extended to the long Weierstrass form (and so to characteristic two and three). One simply changes the definition by replacing reflection in the x-axis by reflection in the line

$$Y = a_1X + a_3.$$

In addition a little calculus will result in explicit algebraic formulae for the chord-tangent process. This is necessary since drawing diagrams as above is not really allowed in a field of finite characteristic. The algebraic formulae are summarized in the following lemma.

LEMMA 2.2. *Let E denote an elliptic curve given by*

$$E : Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$

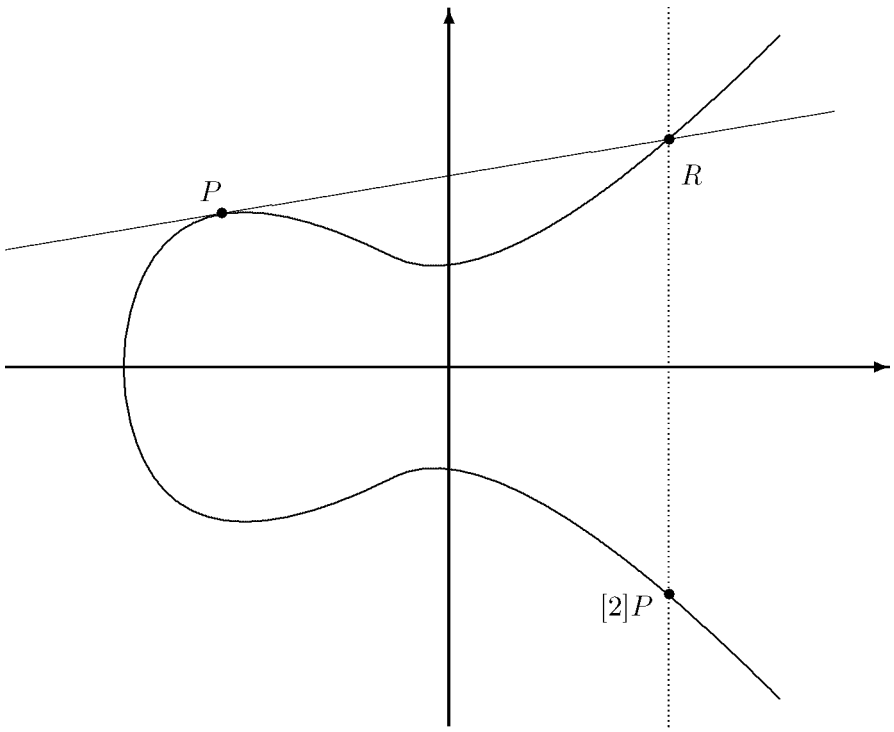
and let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ denote points on the curve. Then

$$-P_1 = (x_1, -y_1 - a_1x_1 - a_3).$$

Set

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1},$$
$$\mu = \frac{y_1x_2 - y_2x_1}{x_2 - x_1}$$

FIGURE 2. Doubling a point on an elliptic curve



when $x_1 \neq x_2$, *and set*

$$\lambda = \frac{3x_1^2 + 2a_2x_1 + a_4 - a_1y_1}{2y_1 + a_1x_1 + a_3},$$
$$\mu = \frac{-x_1^3 + a_4x_1 + 2a_6 - a_3y_1}{2y_1 + a_1x_1 + a_3}$$

when $x_1 = x_2$ *and* $P_2 \neq -P_1$. *If*

$$P_3 = (x_3, y_3) = P_1 + P_2 \neq \mathcal{O}$$

then x_3 *and* y_3 *are given by the formulae*

$$x_3 = \lambda^2 + a_1\lambda - a_2 - x_1 - x_2,$$
$$y_3 = -(\lambda + a_1)x_3 - \mu - a_3.$$

The elliptic curve isomorphisms described earlier then become group isomorphisms as they respect the group structure.

For a positive integer m we let $[m]$ denote the multiplication-by-m map from the curve to itself. This map takes a point P to

$$P + P + \cdots + P,$$

where we have m summands. This map is the basis of elliptic curve cryptography, since whilst it is easy to compute, it is believed to be hard to invert, i.e. given $P = (x, y)$ and $[m]P = (x', y')$ it is

hard to compute m. Of course this statement of hardness assumes a well-chosen elliptic curve etc., something we will return to later in the book.

We end this section with an example of the elliptic curve group law. Again we take our elliptic curve

$$E : Y^2 = X^3 + X + 3$$

over the field $\mathbb{F}_7$. It turns out there are six points on this curve given by

$$\mathcal{O},\ (4,1),\ (6,6),\ (5,0),\ (6,1) \text{ and } (4,6).$$

These form a group with the group law being given by the following table, which is computed using the addition formulae given above.

$+$	$\mathcal{O}$	$(4,1)$	$(6,6)$	$(5,0)$	$(6,1)$	$(4,6)$
$\mathcal{O}$	$\mathcal{O}$	$(4,1)$	$(6,6)$	$(5,0)$	$(6,1)$	$(4,6)$
$(4,1)$	$(4,1)$	$(6,6)$	$(5,0)$	$(6,1)$	$(4,6)$	$\mathcal{O}$
$(6,6)$	$(6,6)$	$(5,0)$	$(6,1)$	$(4,6)$	$\mathcal{O}$	$(4,1)$
$(5,0)$	$(5,0)$	$(6,1)$	$(4,6)$	$\mathcal{O}$	$(4,1)$	$(6,6)$
$(6,1)$	$(6,1)$	$(4,6)$	$\mathcal{O}$	$(4,1)$	$(6,6)$	$(5,0)$
$(4,6)$	$(4,6)$	$\mathcal{O}$	$(4,1)$	$(6,6)$	$(5,0)$	$(6,1)$

As an example of the multiplication-by-m map, if we let

$$P = (4,1)$$

then we have

$$\begin{aligned}
[2]P &= (6,6),\\
[3]P &= (5,0),\\
[4]P &= (6,1),\\
[5]P &= (4,6),\\
[6]P &= \mathcal{O}.
\end{aligned}$$

So we see in this example that $E(\mathbb{F}_7)$ is a finite cyclic abelian group of order six generated by the point P. For all elliptic curves over finite fields the group is always finite and it is also highly likely to be cyclic (or 'nearly' cyclic).

3. Elliptic Curves over Finite Fields

Over a finite field $\mathbb{F}_q$, the number of rational points on a curve is finite, and its size will be denoted by $\#E(\mathbb{F}_q)$. The expected number of points on the curve is around $q+1$ and if we set

$$\#E(\mathbb{F}_q) = q + 1 - t$$

then the value t is called the *trace of Frobenius* at q.

A first approximation to the order of $E(\mathbb{F}_q)$ is given by the following well-known theorem of Hasse.

THEOREM 2.3 (H. Hasse, 1933). *The trace of Frobenius satisfies*

$$|t| \leq 2\sqrt{q}.$$

Consider our example of

$$E : Y^2 = X^3 + X + 3$$

then recall this has six points over the field $\mathbb{F}_7$, and so the associated trace of Frobenius is equal to 2, which is less than $2\sqrt{q} = 2\sqrt{7} = 5.29$.

The qth-power Frobenius map, on an elliptic curve E defined over $\mathbb{F}_q$, is given by

$$\varphi : \begin{cases} E(\overline{\mathbb{F}}_q) \longrightarrow E(\overline{\mathbb{F}}_q) \\ (x,y) \longmapsto (x^q, y^q) \\ \mathcal{O} \longmapsto \mathcal{O}. \end{cases}$$

The map φ sends points on E to points on E, no matter what the field of definition of the point is. In addition the map φ respects the group law in that

$$\varphi(P+Q) = \varphi(P) + \varphi(Q).$$

In other words the map φ is a group endomorphism of E over $\overline{\mathbb{F}}_q$, referred to as the Frobenius endomorphism.

The trace of Frobenius t and the Frobenius endomorphism φ are linked by the equation

$$\varphi^2 - [t]\varphi + [q] = [0].$$

Hence, for any point $P = (x,y)$ on the curve, we have

$$(x^{q^2}, y^{q^2}) - [t](x^q, y^q) + [q](x,y) = \mathcal{O},$$

where addition and subtraction denote curve operations.

There are two particular classes of curves which, under certain conditions, will prove to be cryptographically weak:

- The curve $E(\mathbb{F}_q)$ is said to be anomalous if its trace of Frobenius is one, giving $\#E(\mathbb{F}_q) = q$. These curves are weak when $q = p$, the field characteristic.
- The curve $E(\mathbb{F}_q)$ is said to be supersingular if the characteristic p divides the trace of Frobenius, t. Such curves are usually considered weak cryptographically and are usually avoided. If $q = p$ then this means that $E(\mathbb{F}_p)$ has $p+1$ points since we must have $t = 0$. For other finite fields the possible values of t corresponding to supersingular elliptic curves are given by, where $q = p^f$,
  - f odd: $t = 0$, $t^2 = 2q$ and $t^2 = 3q$.
  - f even : $t^2 = 4q$, $t^2 = q$ if $p = 1 \pmod 3$ and $t = 0$ if $p \neq 1 \pmod 4$.

We will also need to choose a curve such that the group order $\#E(\mathbb{F}_q)$ is divisible by a large prime number. This means we need to be able to compute the group order $\#E(\mathbb{F}_q)$, so as to check both this and the above two conditions.

For any elliptic curve and any finite field the group order $\#E(\mathbb{F}_q)$ can be computed in polynomial time. But this is usually done via a complicated algorithm that we cannot go into in this book. Hence, you should just remember that computing the group order is easy. We shall see in a later chapter, when considering algorithms to solve discrete logarithm problems, that knowing the group order is important in understanding how secure a group is.

One of the advantages of elliptic curves is that there is a very large number of possible groups. One can choose both the finite field and the coefficients of the curve. In addition finding elliptic curves with the correct cryptographic properties to make them secure is relatively easy.

As was apparent from the earlier discussion, the cases char $K = 2, 3$ often require separate treatment. Practical implementations of elliptic curve cryptosystems are usually based on either $\mathbb{F}_{2^n}$, i.e. characteristic two, or $\mathbb{F}_p$ for large primes p. Therefore, in the remainder of this chapter we will focus on fields of characteristic two and $p > 3$, and will omit the separate treatment of the case char $K = 3$. Most arguments, though, carry easily to characteristic three, with modifications that are well documented in the literature.

3.1. Curves over Fields of Characteristic $p > 3$. Assume that our finite field is given by $K = \mathbb{F}_q$, where $q = p^n$ for a prime $p > 3$ and an integer $n \geq 1$. As mentioned, the curve equation in this case can be simplified to the short Weierstrass form

$$E : Y^2 = X^3 + aX + b.$$

The discriminant of the curve then reduces to $\Delta = -16(4a^3 + 27b^2)$, and its j-invariant to $j(E) = -1728(4a)^3/\Delta$. The formulae for the group law in Lemma 2.2 also simplify to

$$-P_1 = (x_1, -y_1),$$

and if

$$P_3 = (x_3, y_3) = P_1 + P_2 \neq \mathcal{O},$$

then x_3 and y_3 are given by the formulae

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2, \\ y_3 &= (x_1 - x_3)\lambda - y_1, \end{aligned}$$

where if $x_1 \neq x_2$ we set

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1},$$

and if $x_1 = x_2$, $y_1 \neq 0$ we set

$$\lambda = \frac{3x_1^2 + a}{2y_1}.$$

3.2. Curves over Fields of Characteristic Two. We now specialize to the case of finite fields where $q = 2^n$ with $n \geq 1$. In this case, the expression for the j-invariant reduces to $j(E) = a_1^{12}/\Delta$. In characteristic two, the condition $j(E) = 0$, i.e. $a_1 = 0$, is equivalent to the curve being supersingular. As mentioned earlier, this very special type of curve is avoided in cryptography. We assume, therefore, that $j(E) \neq 0$.

Under these assumptions, a representative for each isomorphism class of elliptic curves over $\mathbb{F}_q$ is given by

$$E : Y^2 + XY = X^3 + a_2X^2 + a_6, \tag{6}$$

where $a_6 \in \mathbb{F}_q^*$ and $a_2 \in \{0, \gamma\}$ with γ a fixed element in $\mathbb{F}_q$ such that $\mathrm{Tr}_{q|2}(\gamma) = 1$, where $\mathrm{Tr}_{q|2}$ is the absolute trace

$$\mathrm{Tr}_{2^n|2}(\alpha) = \sum_{i=0}^{n-1} \alpha^{2^i}.$$

The formulae for the group law in Lemma 2.2 then simplify to

$$-P_1 = (x_1, y_1 + x_1),$$

and if

$$P_3 = (x_3, y_3) = P_1 + P_2 \neq \mathcal{O},$$

then x_3 and y_3 are given by the formulae

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + a_2 + x_1 + x_2, \\ y_3 &= (\lambda + 1)x_3 + \mu \\ &= (x_1 + x_3)\lambda + x_3 + y_1, \end{aligned}$$

where if $x_1 \neq x_2$ we set

$$\begin{aligned} \lambda &= \frac{y_2 + y_1}{x_2 + x_1}, \\ \mu &= \frac{y_1x_2 + y_2x_1}{x_2 + x_1} \end{aligned}$$

and if $x_1 = x_2 \neq 0$ we set

$$\lambda = \frac{x_1^2 + y_1}{x_1},$$
$$\mu = x_1^2.$$

4. Projective Coordinates

One of the problems with the above formulae for the group laws given in both large and even characteristic is that at some stage they involve a division operation. Division in finite fields is considered as an expensive operation, since it usually involves some variant of the extended Euclidean algorithm, which although of approximately the same complexity as multiplication can usually not be implemented as efficiently.

To avoid these division operations one can use projective coordinates. Here one writes the elliptic curve using three variables (X, Y, Z) instead of just (X, Y). Instead of using the projective representation given at the start of this chapter we instead use one where the curve is written as

$$E : Y^2 + a_1XYZ + a_2YZ^4 = X^3 + a_2X^2Z^2 + a_4XZ^4 + a_6Z^6.$$

The point at infinity is still denoted by $(0, 1, 0)$, but now the map from projective to affine coordinates is given by

$$(X, Y, Z) \longmapsto (X/Z^2, Y/Z^3).$$

This choice of projective coordinates is made to provide a more efficient arithmetic operation.

4.1. Large Prime Characteristic. The formulae for point addition when our elliptic curve is written as

$$Y^2 = X^3 + aXZ^4 + bZ^6$$

are now given by the law

$$(X_3, Y_3, Z_3) = (X_1, Y_1, Z_1) + (X_2, Y_2, Z_2)$$

where (X_3, Y_3, Z_3) are derived from the formulae

$$\begin{aligned}
\lambda_1 &= X_1Z_2^2, & \lambda_2 &= X_2Z_1^2,\\
\lambda_3 &= \lambda_1 - \lambda_2, & \lambda_4 &= Y_1Z_2^3,\\
\lambda_5 &= Y_2Z_1^3, & \lambda_6 &= \lambda_4 - \lambda_5,\\
\lambda_7 &= \lambda_1 + \lambda_2, & \lambda_8 &= \lambda_4 + \lambda_5,\\
Z_3 &= Z_1Z_2\lambda_3, & X_3 &= \lambda_6^2 - \lambda_7\lambda_3^2,\\
\lambda_9 &= \lambda_7\lambda_3^2 - 2X_3, & Y_3 &= (\lambda_9\lambda_6 - \lambda_8\lambda_3^3)/2.
\end{aligned}$$

Notice the avoidance of any division operation, bar division by 2 which can be easily accomplished by multiplication of the precomputed value of $2^{-1} \pmod p$.

Doubling a point,

$$(X_3, Y_3, Z_3) = [2](X_1, Y_1, Z_1),$$

can be accomplished using the formulae

$$\begin{aligned}
\lambda_1 &= 3X_1^2 + aZ_1^4, & Z_3 &= 2Y_1Z_1,\\
\lambda_2 &= 4X_1Y_1^2, & X_3 &= \lambda_1^2 - 2\lambda_2,\\
\lambda_3 &= 8Y_1^4, & Y_3 &= \lambda_1(\lambda_2 - X_3) - \lambda_3.
\end{aligned}$$

4.2. Even Characteristic. In even characteristic we write our elliptic curve in the form

$$Y^2 + XYZ = X^3 + a_2X^2Z^2 + a_6Z^6.$$

Point addition,

$$(X_3, Y_3, Z_3) = (X_1, Y_1, Z_1) + (X_2, Y_2, Z_2)$$

is now accomplished using the recipe

$$\begin{aligned}
\lambda_1 &= X_1Z_2^2, & \lambda_2 &= X_2Z_1^2,\\
\lambda_3 &= \lambda_1 + \lambda_2, & \lambda_4 &= Y_1Z_2^3,\\
\lambda_5 &= Y_2Z_1^3, & \lambda_6 &= \lambda_4 + \lambda_5,\\
\lambda_7 &= Z_1\lambda_3, & \lambda_8 &= \lambda_6X_2 + \lambda_7Y_2,\\
Z_3 &= \lambda_7Z_2, & \lambda_9 &= \lambda_6 + Z_3,\\
X_3 &= a_2Z_3^2 + \lambda_6\lambda_9 + \lambda_3^3, & Y_3 &= \lambda_9X_3 + \lambda_8\lambda_7^2.
\end{aligned}$$

Doubling is then performed using

$$\begin{aligned}
Z_3 &= X_1Z_1^2, & X_3 &= (X_1 + d_6Z_1^2)^4,\\
\lambda &= Z_3 + X_1^2 + Y_1Z_1, & Y_3 &= X_1^4Z_3 + \lambda X_3,
\end{aligned}$$

where $d_6 = \sqrt[4]{a_6}$.

Notice how in both even and odd characteristic we have avoided a division operation.

5. Point Compression

In many cryptographic protocols we need to store or transmit an elliptic curve point. Using affine coordinates this can be accomplished using two field elements, i.e. by transmitting x and then y. However, one can do better using a technique called point compression.

Point compression works by the observation that for every x-coordinate on the curve there are at most two corresponding y-coordinates. Hence, we can represent a point by storing the x-coordinate along with a bit b to say which value of the y-coordinate we should take. All that remains to decide is how to compute the bit b and how to reconstruct the y-coordinate given the x-coordinate and the bit b.

5.1. Large Prime Characteristic. For elliptic curves over fields of large prime characteristic we notice that if $\alpha \in \mathbb{F}_p^*$ then the two square roots $\pm\beta$ of α have different parities, when represented as integers in the range $[1, \ldots, p-1]$. This is because

$$-\beta = p - \beta.$$

Hence, as the bit b we choose the parity of the y-coordinate.

Then, given (x, b), we can reconstruct y by computing

$$\beta = \sqrt{x^3 + ax + b} \pmod{p}.$$

If the parity of β is equal to b we set $y = \beta$, otherwise we set $y = p - \beta$. If $\beta = 0$ then no matter which value of b we have we set $y = 0$.

5.2. Even Characteristic. In even characteristic we need to be slightly more clever. Suppose we are given a point $P = (x, y)$ on the elliptic curve

$$Y^2 + XY = X^3 + a_2X + a_6.$$

If $y = 0$ then we set $b = 0$, otherwise we compute

$$z = y/x$$

and let b denote the least significant bit of z. To recover y given (x, b), for $x \neq 0$, we set

$$\alpha = x + a_2 + \frac{a_6}{x^2}$$

and let β denote a solution of

$$z^2 + z = \alpha.$$

Then if the least significant bit of β is equal to b we set $y = x\beta$, otherwise we set $y = x(\beta + 1)$. To see why this works notice that if (x, y) is a solution of

$$Y^2 + XY = X^3 + a_2X^2 + a_6$$

then $(x, y/x)$ and $(x, 1 + y/x)$ are the two solutions of

$$Z^2 + Z = X + a_2 + \frac{a_6}{X^2}.$$

As an example consider the curve

$$E : Y^2 = X^3 + X + 3$$

over the field $\mathbb{F}_7$. Then the points $(4, 1)$ and $(4, 6)$ which in bits we need to represent as

$$(0b100, 0b001) \text{ and } (0b100, 0b110),$$

i.e. requiring six bits for each point, can be represented as

$$(0b100, 0b1) \text{ and } (0b100, 0b0),$$

where we only use four bits for each point.

In larger, cryptographically interesting, examples the advantage becomes more pronounced. For example consider the same curve over the field

$$p = 1\,125\,899\,906\,842\,679 = 2^{50} + 55$$

then the point

$$(1\,125\,899\,906\,842\,675, 245\,132\,605\,757\,739)$$

can be represented by the integers

$$(1\,125\,899\,906\,842\,675, 1).$$

So instead of requiring 102 bits we only require 52 bits.

Chapter Summary

- Elliptic curves over finite fields are another example of a finite abelian group. There are a lot of such groups since we are free to choose both the curve and the field.
- For cryptography we need to be able to compute the number of elements in the group. Although this is done using a complicated algorithm, it can be done in polynomial time.

- One should usually avoid supersingular and anomalous curves in cryptographic applications.
- Efficient algorithms for the group law can be produced by using projective coordinates. These algorithms avoid the need for costly division operations in the underlying finite field.
- To save bandwidth and space it is possible to efficiently compress elliptic curve points (x, y) down to x and a single bit b. The uncompression can also be performed efficiently.

Further Reading

For those who wish to learn more about elliptic curves in general try the textbook by Silverman (which is really aimed at mathematics graduate students). For those who are simply interested in the cryptographic applications of elliptic curves and the associated algorithms and techniques see the book by Blake, Seroussi and Smart.

I.F. Blake, G. Seroussi and N.P. Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, 1999.

J.H. Silverman. *The Arithmetic of Elliptic Curves.* Springer-Verlag, 1985.

Part 2

Symmetric Encryption

Encryption of most data is accomplished using fast block and stream ciphers. These are examples of symmetric encryption algorithms. In addition all historical, i.e. pre-1960, ciphers are symmetric in nature and share some design principles with modern ciphers.

The main drawback with symmetric ciphers is that they give rise to a problem of how to distribute the secret keys between users, so we also address this issue.

We also discuss the properties and design of cryptographic hash functions and message authentication codes. Both of which will form basic building blocks of other schemes and protocols within this book.

In the following chapters we explain the theory and practice of modern symmetric ciphers, but first we consider historical ciphers.

CHAPTER 3

Historical Ciphers

Chapter Goals

- To explain a number of historical ciphers, such as the Caesar cipher, substitution cipher.
- To show how these historical ciphers can be broken because they do not hide the underlying statistics of the plaintext.
- To introduce the concepts of substitution and permutation as basic cipher components.
- To introduce a number of attack techniques, such as chosen plaintext attacks.

1. Introduction

An encryption algorithm, or cipher, is a means of transforming plaintext into ciphertext under the control of a secret key. This process is called encryption or encipherment. We write

$$c = e_k(m),$$

where

- m is the plaintext,
- e is the cipher function,
- k is the secret key,
- c is the ciphertext.

The reverse process is called decryption or decipherment, and we write

$$m = d_k(c).$$

Note, that the encryption and decryption algorithms e, d are public, the secrecy of m given c depends totally on the secrecy of k.

The above process requires that each party needs access to the secret key. This needs to be known to both sides, but needs to be kept secret. Encryption algorithms which have this property are called *symmetric cryptosystems* or secret key cryptosystems. There is a form of cryptography which uses two different types of key, one is publicly available and used for encryption whilst the other is private and used for decryption. These latter types of cryptosystems are called *asymmetric cryptosystems* or *public key cryptosystems*, to which we shall return in a later chapter.

Usually in cryptography the communicating parties are denoted by A and B. However, often one uses the more user-friendly names of Alice and Bob. But you should not assume that the parties are necessarily human, we could be describing a communication being carried out between two autonomous machines. The eavesdropper, bad girl, adversary or attacker is usually given the name Eve.

In this chapter we shall present some historical ciphers which were used in the pre-computer age to encrypt data. We shall show that these ciphers are easy to break as soon as one understands the statistics of the underlying language, in our case English. In Chapter 5 we shall study this relationship between how easy the cipher is to break and the statistical distribution of the underlying plaintext.

TABLE 1. English letter frequencies

Letter	Percentage	Letter	Percentage
A	8.2	N	6.7
B	1.5	O	7.5
C	2.8	P	1.9
D	4.2	Q	0.1
E	12.7	R	6.0
F	2.2	S	6.3
G	2.0	T	9.0
H	6.1	U	2.8
I	7.0	V	1.0
J	0.1	W	2.4
K	0.8	X	0.1
L	4.0	Y	2.0
M	2.4	Z	0.1

FIGURE 1. English letter frequencies

The distribution of English letter frequencies is described in Table 1, or graphically in Fig. 1. As one can see the most common letters are E and T. It often helps to know second order statistics about the underlying language, such as which are the most common sequences of two or three letters, called bigrams and trigrams. The most common bigrams in English are given by Table 2, with the associated approximate percentages. The most common trigrams are, in decreasing order,

THE, ING, AND, HER, ERE, ENT, THA, NTH, WAS, ETH, FOR.

Armed with this information about English we are now able to examine and break a number of historical ciphers.

TABLE 2. English bigram frequencies

Bigram	Percentage	Bigram	Percentage
TH	3.15	HE	2.51
AN	1.72	IN	1.69
ER	1.54	RE	1.48
ES	1.45	ON	1.45
EA	1.31	TI	1.28
AT	1.24	ST	1.21
EN	1.20	ND	1.18

2. Shift Cipher

We first present one of the earliest ciphers, called the shift cipher. Encryption is performed by replacing each letter by the letter a certain number of places on in the alphabet. So for example if the key was three, then the plaintext A would be replaced by the ciphertext D, the letter B would be replaced by E and so on. The plaintext word HELLO would be encrypted as the ciphertext KHOOR. When this cipher is used with the key three, it is often called the Caesar cipher, although in many books the name Caesar cipher is sometimes given to the shift cipher with any key. Strictly this is not correct since we only have evidence that Julius Caesar used the cipher with the key three.

There is a more mathematical explanation of the shift cipher which will be instructive for future discussions. First we need to identify each letter of the alphabet with a number. It is usual to identify the letter A with the number 0, the letter B with number 1, the letter C with the number 2 and so on until we identify the letter Z with the number 25. After we convert our plaintext message into a sequence of numbers, the ciphertext in the shift cipher is obtained by adding to each number the secret key k modulo 26, where the key is a number in the range 0 to 25. In this way we can interpret the shift cipher as a *stream cipher*, with key stream given by the repeating sequence

$$k, k, k, k, k, k, \ldots$$

This key stream is not very random, which results in it being easy to break the shift cipher. A naive way of breaking the shift cipher is to simply try each of the possible keys in turn, until the correct one is found. There are only 26 possible keys so the time for this exhaustive key search is very small, particularly if it is easy to recognize the underlying plaintext when it is decrypted.

We shall show how to break the shift cipher by using the statistics of the underlying language. Whilst this is not strictly necessary for breaking this cipher, later we shall see a cipher that is made up of a number of shift ciphers applied in turn and then the following statistical technique will be useful. Using a statistical technique on the shift cipher is also instructive as to how statistics of the underlying plaintext can arise in the resulting ciphertext.

Take the following example ciphertext, which since it is public knowledge we represent in blue.

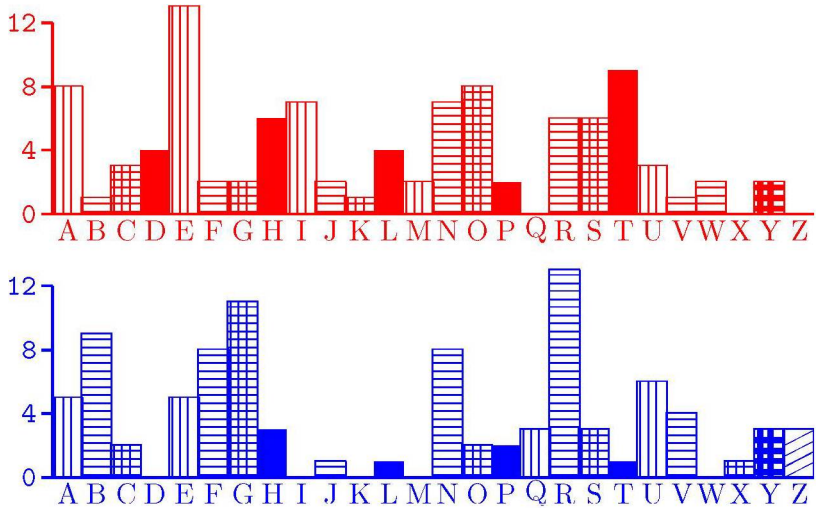
GB OR, BE ABG GB OR: GUNG VF GUR DHRFGVBA:
JURGURE 'GVF ABOYRE VA GUR ZVAQ GB FHSSRE
GUR FYVATF NAQ NEEBJF BS BHGENTRBHF SBEGHAR,
BE GB GNXR NEZF NTNVAFG N FRN BS GEBHOYRF,
NAQ OL BCCBFVAT RAQ GURZ? GB QVR: GB FYRRC;
AB ZBER; NAQ OL N FYRRC GB FNL JR RAQ
GUR URNEG-NPUR NAQ GUR GUBHFNAQ ANGHENY FUBPXF
GUNG SYRFU VF URVE GB, 'GVF N PBAFHZZNGVBA
QRIBHGYL GB OR JVFU'Q. GB QVR, GB FYRRC;
GB FYRRC: CREPUNAPR GB QERNZ: NL, GURER'F GUR EHO;
SBE VA GUNG FYRRC BS QRNGU JUNG QERNZF ZNL PBZR
JURA JR UNIR FUHSSYRQ BSS GUVF ZBEGNY PBVY,
ZHFG TVIR HF CNHFR: GURER'F GUR ERFCRPG
GUNG ZNXRF PNYNZVGL BS FB YBAT YVSR;

One technique of breaking the previous sample ciphertext is to notice that the ciphertext still retains details about the word lengths of the underlying plaintext. For example the ciphertext letter N appears as a single letter word. Since the only single letter words in English are A and I we can conclude that the key is either 13, since N is thirteen letters on from A in the alphabet, or the key is equal to 5, since N is five letters on from I in the alphabet. Hence, the moral here is to always remove word breaks from the underlying plaintext before encrypting using the shift

cipher. But even if we ignore this information about the words we can still break this cipher using frequency analysis.

We compute the frequencies of the letters in the ciphertext and compare them with the frequencies obtained from English which we saw in Fig. 1. We present the two bar graphs one above each other in Fig. 2 so you can see that one graph looks almost like a shift of the other graph. The statistics obtained from the sample ciphertext are given in blue, whilst the statistics obtained from the underlying plaintext language are given in red. Note, we do not compute the red statistics from the actual plaintext since we do not know this yet, we only make use of the knowledge of the underlying language.

FIGURE 2. Comparison of plaintext and ciphertext frequencies for the shift cipher example



By comparing the two bar graphs in Fig. 2 we can see by how much we think the blue graph has been shifted compared with the red graph. By examining where we think the plaintext letter E may have been shifted, one can hazard a guess that it is shifted by one of

$$2, 9, 13 \text{ or } 23.$$

Then by trying to deduce by how much the plaintext letter A has been shifted we can guess that it has been shifted by one of

$$1, 6, 13 \text{ or } 17.$$

The only shift value which is consistent appears to be the value 13, and we conclude that this is the most likely key value. We can now decrypt the ciphertext, using this key. This reveals, that the underlying plaintext is:

To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them? To die: to sleep;
No more; and by a sleep to say we end
The heart-ache and the thousand natural shocks
That flesh is heir to, 'tis a consummation
Devoutly to be wish'd. To die, to sleep;
To sleep: perchance to dream: ay, there's the rub;
For in that sleep of death what dreams may come

When we have shuffled off this mortal coil,
Must give us pause: there's the respect
That makes calamity of so long life;

The above text is obviously taken from *Hamlet* by William Shakespeare.

3. Substitution Cipher

The main problem with the shift cipher is that the number of keys is too small, we only have 26 possible keys. To increase the number of keys a *substitution cipher* was invented. To write down a key for the substitution cipher we first write down the alphabet, and then a permutation of the alphabet directly below it. This mapping gives the substitution we make between the plaintext and the ciphertext

Plaintext alphabet	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ciphertext alphabet	GOYDSIPELUAVCRJWXZNHBQFTMK

Encryption involves replacing each letter in the top row by its value in the bottom row. Decryption involves first looking for the letter in the bottom row and then seeing which letter in the top row maps to it. Hence, the plaintext word HELLO would encrypt to the ciphertext ESVVJ if we used the substitution given above.

The number of possible keys is equal to the total number of permutations on 26 letters, namely the size of the group S_{26}, which is

$$26! \approx 4.03 \cdot 10^{26} \approx 2^{88}.$$

Since, as a rule of thumb, it is feasible to only run a computer on a problem which takes under 2^{80} steps we can deduce that this large key space is far too large to enable a brute force search even using a modern computer. Still we can break substitution ciphers using statistics of the underlying plaintext language, just as we did for the shift cipher.

Whilst the shift cipher can be considered as a stream cipher since the ciphertext is obtained from the plaintext by combining it with a keystream, the substitution cipher operates much more like a modern block cipher, with a block length of one English letter. A ciphertext block is obtained from a plaintext block by applying some (admittedly simple) key dependent algorithm.

Substitution ciphers are the types of ciphers commonly encountered in puzzle books, they have an interesting history and have occurred in literature. See for example the Sherlock Holmes story *The Adventure of the Dancing Men* by Arthur Conan–Doyle. The plot of this story rests on a substitution cipher where the ciphertext characters are taken from an alphabet of 'stick men' in various positions. The method of breaking the cipher as described by Holmes to Watson in this story is precisely the method we shall adopt below.

We give a detailed example, which we make slightly easier by keeping in the ciphertext details about the underlying word spacing used in the plaintext. This is only for ease of exposition, the techniques we describe can still be used if we ignore these word spacings, although more care and thought is required.

Consider the ciphertext

XSO MJIWXVL JODIVA STW VAO VY OZJVCO'W LTJDOWX KVAKOAXJTXIVAW VY SIDS XOKSAVLVDQ IAGZWXJQ. KVUCZXOJW, KVUUZAIKTXIVAW TAG UIKJVOLOKXJVAIKW TJO HOLL JOCJOWOAXOG, TLVADWIGO GIDIXTL UOGIT, KVUCZXOJ DTUOW TAG OLOKXJVAIK KVUUOJKO. TW HOLL TW SVWXIAD UTAQ JOWOTJKS TAG CJVGZKX GONOLVCUOAX KOAXJOW VY UTPVJ DLVMTL KVUCTAIOW, XSO JODIVA STW T JTCIGLQ DJVHIAD AZUMOJ VY IAAVNTXINO AOH KVUCTAIOW. XSO KVUCZXOJ WKIOAKO GOCTJXUOAX STW KLVWO JOLTXIVAWSICW HIXS UTAQ VY XSOWO VJDTAIWTXIVAW NIT KVLLTMVJTXINO CJVPOKXW, WXTYY WOKVAGUOAXW TAG NIWIXIAD IAGZWXJITL WXTYY. IX STW JOKOAXLQ IAXJVGZKOG

WONOJTL UOKSTAIWUW YVJ GONOLVCIAD TAG WZCCVJXIAD OAXJOCJOAOZJITL WXZGOAXW TAG WXTYY, TAG TIUW XV CLTQ T WIDAIYIKTAX JVLO IA XSO GONOLVCUOAX VY SIDS-XOKSAVLVDQ IAGZWXJQ IA XSO JODIVA.

XSO GOCTJXUOAX STW T LTJDO CJVDJTUUO VY JOWOTJKS WZCCVJXOG MQ IAGZWXJQ, XSO OZJVCOTA ZAIVA, TAG ZE DVNOJAUOAX JOWOTJKS OWXTMLIW-SUOAXW TAG CZMLIK KVJCVJTXIVAW. T EOQ OLOUOAX VY XSIW IW XSO WXJ-VAD LIAEW XSTX XSO GOCTJXUOAX STW HIXS XSO KVUCZXOJ, KVUUZAIKTXIVAW, UIKJVOLOKXJVAIKW TAG UOGIT IAGZWXJIOW IA XSO MJIWXVL JODIVA . XSO TKT-GOUIK JOWOTJKS CJVDJTUUO IW VJDTAIWOG IAXV WONOA DJVZCW, LTADZTDOW TAG TJKSIXOKXZJO, GIDIXTL UOGIT, UVMILO TAG HOTJTMLO KVUCZXIAD, UTK-SIAO LOTJAIAD, RZTAXZU KVUCZXIAD, WQWXOU NOJIYIKTXIVA, TAG KJQCXVD-JTCSQ TAG IAYVJUTXIVA WOKZJIXQ.

We can compute the following frequencies for single letters in the above ciphertext:

Letter	Freq	Letter	Freq	Letter	Freq
A	8.6995	B	0.0000	C	3.0493
D	3.1390	E	0.2690	F	0.0000
G	3.6771	H	0.6278	I	7.8923
J	7.0852	K	4.6636	L	3.5874
M	0.8968	N	1.0762	O	11.479
P	0.1793	Q	1.3452	R	0.0896
S	3.5874	T	8.0717	U	4.1255
V	7.2645	W	6.6367	X	8.0717
Y	1.6143	Z	2.7802		

In addition we determine that the most common bigrams in this piece of ciphertext are

TA, AX, IA, VA, WX, XS, AG, OA, JO, JV,

whilst the most common trigrams are

OAX, TAG, IVA, XSO, KVU, TXI, UOA, AXS.

Since the ciphertext letter O occurs with the greatest frequency, namely 11.479, we can guess that the ciphertext letter O corresponds to the plaintext letter E. We now look at what this means for two of the common trigrams found in the ciphertext

- The ciphertext trigram OAX corresponds to E * *.
- The ciphertext trigram XSO corresponds to * * E.

We examine similar common similar trigrams in English, which start or end with the letter E. We find that three common ones are given by ENT, ETH and THE. Since the two trigrams we wish to match have one starting with the same letter as the other finishes with, we can conclude that it is highly likely that we have the correspondence

- X = T,
- S = H,
- A = N.

Even after this small piece of analysis we find that it is much easier to understand what the underlying plaintext should be. If we focus on the first two sentences of the ciphertext we are trying to break, and we change the letters which we think we have found the correct mappings for, then we obtain:

THE MJIWTVL JEDIVN HTW VNE VY EZJVCE'W LTJDEWT
KVNKENTJTTIV NW VY HIDH TEKHNVLVDQ INGZWTJQ.
KVUCZTEJW, KVUUZNIKTTIVNW TNG UIKJVELEKTJVNIKW

TJE HELL JECJEWENTEG, TLVNDWIGE GIDITTL UEGIT,
KVUCZTEJ DTUEW TNG ELEKTJVNIK KVUUEJKE.

Recall, this was after the four substitutions

$$O = E,\ X = T,\ S = H,\ A = N.$$

We now cheat and use the fact that we have retained the word sizes in the ciphertext. We see that since the letter T occurs as a single ciphertext letter we must have

$$T = I \text{ or } T = A.$$

The ciphertext letter T occurs with a probability of 8.0717, which is the highest probability left, hence we are far more likely to have

$$T = A.$$

We have already considered the most popular trigram in the ciphertext so turning our attention to the next most popular trigram we see that it is equal to TAG which we suspect corresponds to the plaintext AN*. Therefore it is highly likely that G = D, since AND is a popular trigram in English.

Our partially decrypted ciphertext is now equal to

THE MJIWTVL JEDIVN HAW VNE VY EZJVCE'W LAJDEWT
KVNKENTJATIV NW VY HIDH TEKHNVLVDQ INDZWTJQ.
KVUCZTEJW, KVUUZNIKATIVNW AND UIKJVELEKTJVNIKW
AJE HELL JECJEWENTED, ALVNDWIDE DIDITAL UEDIA,
KVUCZTEJ DAUEW AND ELEKTJVNIK KVUUEJKE.

This was after the six substitutions

$$O = E,\ X = T,\ S = H,$$
$$A = N,\ T = A,\ G = D.$$

We now look at two-letter words which occur in the ciphertext:

- IX
  This corresponds to the plaintext *T. Therefore the ciphertext letter I must be one of the plaintext letters A or I, since the only two-letter words in English ending in T are AT and IT. We already have worked out what the plaintext character A corresponds to, hence we must have I = I.
- XV
  This corresponds to the plaintext T*. Hence, we must have V = O.
- VY
  This corresponds to the plaintext O*. Hence, the ciphertext letter Y must correspond to one of F, N or R. We already know the ciphertext letter corresponding to N. In the ciphertext the probability of Y occurring is 1.6, but in English we expect F to occur with probability 2.2 and R to occur with probability 6.0. Hence, it is more likely that Y = F.
- IW
  This corresponds to the plaintext I*. Therefore, the plaintext character W must be one of F, N, S and T. We already have F, N, T, hence W = S.

All these deductions leave the partial ciphertext as

THE MJISTOL JEDION HAS ONE OF EZJOCE'S LAJDEST
KONKENTJATIONS OF HIDH TEKHNOLODQ INDZSTJQ.
KOUCZTEJS, KOUUZNIKATIONS AND UIKJOELEKTJONIKS AJE
HELL JECJESENTED, ALONDSIDE DIDITAL UEDIA,
KOUCZTEJ DAUES AND ELEKTJONIK KOUUEJKE.

This was after the ten substitutions

$$O = E,\ X = T,\ S = H,\ A = N,\ T = A,$$
$$G = D,\ I = I,\ V = O,\ Y = F,\ W = S.$$

Even with half the ciphertext letters determined it is now quite easy to understand the underlying plaintext, taken from the website of the University of Bristol Computer Science Department. We leave it to the reader to determine the final substitutions and recover the plaintext completely.

4. Vigenère Cipher

The problem with the shift cipher and the substitution cipher was that each plaintext letter always encrypted to the same ciphertext letter. Hence underlying statistics of the language could be used to break the cipher. For example it was easy to determine which ciphertext letter corresponded to the plaintext letter E. From the early 1800s onwards, cipher designers tried to break this link between the plaintext and ciphertext.

The substitution cipher we used above was a mono-alphabetic substitution cipher, in that only one alphabet substitution was used to encrypt the whole alphabet. One way to solve our problem is to take a number of substitution alphabets and then encrypt each letter with a different alphabet. Such a system is called a polyalphabetic substitution cipher.

For example we could take

Plaintext alphabet	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ciphertext alphabet one	TMKGOYDSIPELUAVCRJWXZNHBQF
Ciphertext alphabet two	DCBAHGFEMLKJIZYXWVUTSRQPON

Then the plaintext letters in an odd position we encrypt using the first ciphertext alphabet, whilst the plaintext letters in even positions we encrypt using the second alphabet. For example the plaintext word HELLO, using the above alphabets would encrypt to SHLJV. Notice that the two occurrences of L in the plaintext encrypt to two different ciphertext characters. Thus we have made it harder to use the underlying statistics of the language. If one now does a naive frequency analysis we no longer get a common ciphertext letter corresponding to the plaintext letter E.

We essentially are encrypting the message two letters at a time, hence we have a block cipher with block length two English characters. In real life one may wish to use around five rather than just two alphabets and the resulting key becomes very large indeed. With five alphabets the total key space is

$$(26!)^5 \approx 2^{441},$$

but the user only needs to remember the key which is a sequence of

$$26 \cdot 5 = 130$$

letters. However, just to make life hard for the attacker, the number of alphabets in use should also be hidden from his view and form part of the key. But for the average user in the early 1800s this was far too unwieldy a system, since the key was too hard to remember.

Despite its shortcomings the most famous cipher during the 19th-century was based on precisely this principle. The *Vigenère cipher*, invented in 1533 by Giovan Batista Belaso, was a variant on the above theme, but the key was easy to remember. When looked at in one way the Vigenère cipher is a polyalphabetic block cipher, but when looked at in another, it is a stream cipher which is a natural generalization of the shift cipher.

The description of the Vigenère cipher as a block cipher takes the description of the polyalphabetic cipher above but restricts the possible plaintext alphabets to one of the 26 possible cyclic shifts of the standard alphabet. Suppose five alphabets were used, this reduces the key space down to

$$26^5 \approx 2^{23}$$

and the size of the key to be remembered as a sequence of five numbers between 0 and 25.

However, the description of the Vigenère cipher as a stream cipher is much more natural. Just like the shift cipher, the Vigenère cipher again identifies letters with the numbers $0, \ldots, 25$. The secret key is a short sequence of letters (e.g. a word) which is repeated again and again to form

a keystream. Encryption involves adding the plaintext letter to a key letter. Thus if the key is SESAME, encryption works as follows,

```
THISISATESTMESSAGE
SESAMESESAMESESAME
LLASUWSXWSFQWWKASI
```

Again we notice that A will encrypt to a different letter depending on where it appears in the message.

But the Vigenère cipher is still easy to break using the underlying statistics of English. Once we have found the length of the keyword, breaking the ciphertext is the same as breaking the shift cipher a number of times.

As an example, suppose the ciphertext is given by

UTPDHUG NYH USVKCG MVCE FXL KQIB. WX RKU GI TZN, RLS BBHZLXMSNP KDKS; CEB IH HKEW IBA, YYM SBR PFR SBS, JV UPL O UVADGR HRRWXF. JV ZTVOOV YH ZCQU Y UKWGEB, PL UQFB P FOUKCG, TBF RQ VHCF R KPG, OU KFT ZCQU MAW QKKW ZGSY, FP PGM QKFTK UQFB DER EZRN, MCYE, MG UCTFSVA, WP KFT ZCQU MAW KQIJS. LCOV NTHDNV JPNUJVB IH GGV RWX ONKCGTHKFL XG VKD, ZJM VG CCI MVGD JPNUJ, RLS EWVKJT ASGUCS MVGD; DDK VG NYH PWUV CCHIIY RD DBQN RWTH PFRWBBI VTTK VCGNTGSF FL IAWU XJDUS, HFP VHCF, RR LAWEY QDFS RVMEES FZB CHH JRTT MVGZP UBZN FD ATIIYRTK WP KFT HIVJCI; TBF BLDPWPX RWTH ULAW TG VYCHX KQLJS US DCGCW OPPUPR, VG KFDNUJK GI JIKKC PL KGCJ IAOV KFTR GJFSAW KTZLZES WG RWXWT VWTL WP XPXGG, CJ FPOS VYC BTZCUW XG ZGJQ PMHTRAIBJG WMGFG. JZQ DPB JVYGM ZCLEWXR: CEB IAOV NYH JIKKC TGCWXF UHF JZK.

WX VCU LD YITKFTK WPKCGVCWIQT PWVY QEBFKKQ, QNH NZTTW IRFL IAS VFRPE ODJRXGSPTC EKWPTGEES, GMCG
TTVVPLTFFJ; YCW WV NYH TZYRWH LOKU MU AWO, KFPM VG BLTP VQN RD DSGG AWKWUKKPL KGCJ, XY OPP KPG ONZTT ICUJCHLSF KFT DBQNJTWUG. DYN MVCK ZT MFWCW HTWF FD JL, OPU YAE CH LQ! PGR UF, YH MWPP RXF CDJCGOSF, XMS UZGJQ JL, SXVPN HBG!

There is a way of finding the length of the keyword, which is repeated to form the keystream, called the *Kasiski test*. First we need to look for repeated sequences of characters. Recall that English has a large repetition of certain bigrams or trigrams and over a long enough string of text these are likely to match up to the same two or three letters in the key every so often. By examining the distance between two repeated sequences we can guess the length of the keyword. Each of these distances should be a multiple of the keyword, hence taking the greatest common divisor of all distances between the repeated sequences should give a good guess as to the keyword length.

Let us examine the above ciphertext and look for the bigram WX. The gaps between some of the occurrences of this bigram are 9, 21, 66 and 30, some of which may have occurred by chance, whilst some may reveal information about the length of the keyword. We now take the relevant greatest common divisors to find,

$$\begin{aligned}\gcd(30,66) &= 6,\\ \gcd(3,9) = \gcd(9,66) &= \gcd(9,30) = \gcd(21,66) = 3.\end{aligned}$$

We are unlikely to have a keyword of length three so we conclude that the gaps of 9 and 21 occurred purely by chance. Hence, our best guess for the keyword is that it is of length 6.

Now we take every sixth letter and look at the statistics just as we did for a shift cipher to deduce the first letter of the keyword. We can now see the advantage of using the histograms to break the shift cipher earlier. If we used the naive method and tried each of the 26 keys in turn we

could still not detect which key is correct, since every sixth letter of an English sentence does not produce an English sentence. Using our earlier histogram based method is more efficient in this case.

FIGURE 3. Comparison of plaintext and ciphertext frequencies for every sixth letter of the Vigenère example, starting with the first letter

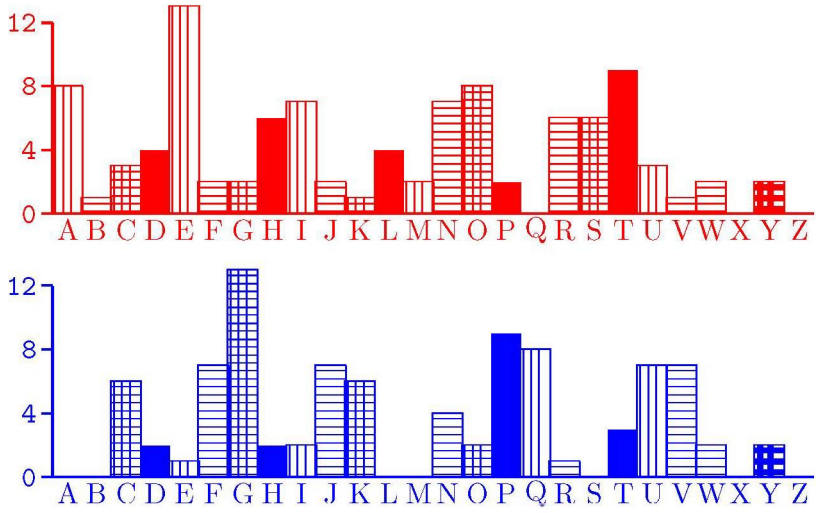
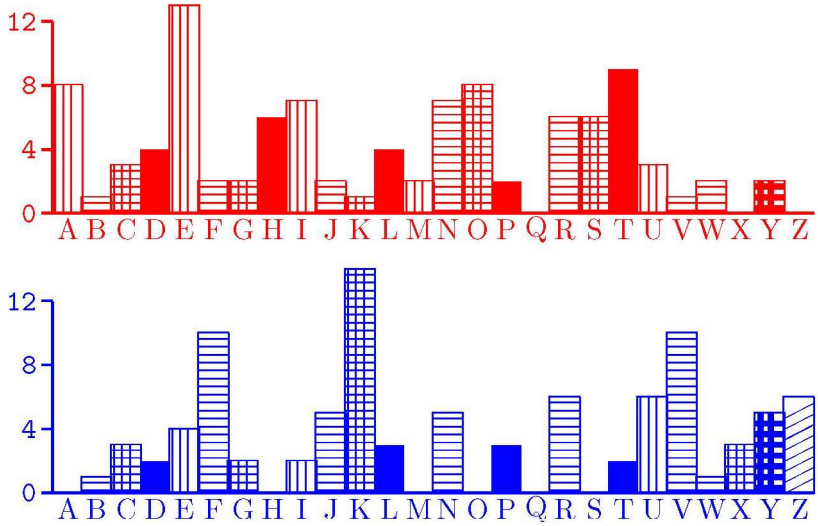


FIGURE 4. Comparison of plaintext and ciphertext frequencies for every sixth letter of the Vigenère example, starting with the second letter



The relevant bar charts for every sixth letter starting with the first are given in Fig. 3. We look for the possible locations of the three peaks corresponding to the plaintext letters A, E and T. We see that this sequence seems to be shifted by two positions in the blue graph compared with the red graph. Hence we can conclude that the first letter of the keyword is C, since C corresponds to a shift of two.

We perform a similar step for every sixth letter, starting with the second one. The resulting bar graphs are given in Fig. 4. Using the same technique we find that the blue graph appears to

have been shifted along by 17 spaces, which corresponds to the second letter of the keyword being equal to R.

Continuing in a similar way for the remaining four letters of the keyword we find the keyword is

CRYPTO.

The underlying plaintext is then found to be:

Scrooge was better than his word. He did it all, and infinitely more; and to Tiny Tim, who did not die, he was a second father. He became as good a friend, as good a master, and as good a man, as the good old city knew, or any other good old city, town, or borough, in the good old world. Some people laughed to see the alteration in him, but he let them laugh, and little heeded them; for he was wise enough to know that nothing ever happened on this globe, for good, at which some people did not have their fill of laughter in the outset; and knowing that such as these would be blind anyway, he thought it quite as well that they should wrinkle up their eyes in grins, as have the malady in less attractive forms. His own heart laughed: and that was quite enough for him.

He had no further intercourse with Spirits, but lived upon the Total Abstinence Principle, ever afterwards; and it was always said of him, that he knew how to keep Christmas well, if any man alive possessed the knowledge. May that be truly said of us, and all of us! And so, as Tiny Tim observed, God bless Us, Every One!

The above text is taken from *A Christmas Carol* by Charles Dickens.

5. A Permutation Cipher

The ideas behind substitution type ciphers forms part of the design of modern symmetric systems. For example later we shall see that both DES and Rijndael make use of a component called an S-Box, which is simply a substitution. The other component that is used in modern symmetric ciphers is based on permutations.

Permutation ciphers have been around for a number of centuries. Here we shall describe the simplest, which is particularly easy to break. We first fix a permutation group S_n and a permutation

$$\sigma \in S_n.$$

It is the value of σ which will be the secret key. As an example suppose we take

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 1 & 3 & 5 \end{pmatrix} = (1243) \in S_5.$$

Now take some plaintext, say

Once upon a time there was a little girl called snow white.

We break the text into chunks of 5 letters

onceu ponat imeth erewa salit tlegi rlcal ledsn owwhi te.

We first pad the message, with some random letters, so that we have a multiple of five letters in each chunk.

onceu ponat imeth erewa salit tlegi rlcal ledsn owwhi teahb.

Then we take each five-letter chunk in turn and swap the letters around according to our secret permutation σ. With our example we obtain

coenu npaot eitmh eewra lsiat etgli crall dlsdn wohwi atheb.

We then remove the spaces, so as to hide the value of n, producing the ciphertext

coenunpaoteitmheewralsiatetglicralldlsdnwohwiatheb.

However, breaking a permutation cipher is easy with a chosen plaintext attack, assuming the group of permutations used (i.e. the value of n) is reasonably small. To attack this cipher we mount a chosen plaintext attack, and ask one of the parties to encrypt the message

abcdefghijklmnopqrstuvwxyz,

to obtain the ciphertext

cadbehfigjmknlorpsqtwuxvyz.

We can then deduce that the permutation looks something like

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & \dots \\ 2 & 4 & 1 & 3 & 5 & 7 & 9 & 6 & 8 & 10 & 12 & 14 & 11 & 13 & 15 & \dots \end{pmatrix}.$$

We see that the sequence repeats (modulo 5) after every five steps and so the value of n is probably equal to five. We can recover the key by simply taking the first five columns of the above permutation.

Chapter Summary

- Many early ciphers can be broken because they do not successfully hide the underlying statistics of the language.
- Important principles behind early ciphers are those of substitution and permutation.
- Ciphers can either work on blocks of characters via some keyed algorithm or simply consist of adding some keystream to each plaintext character.
- Ciphers which aimed to get around these early problems often turned out to be weaker than expected, either due to some design flaw or due to bad key management practices adopted by operators.

Further Reading

The best book on the history of ciphers is that by Kahn. Kahn's book is a weighty tome so those wishing a more rapid introduction should consult the book by Singh. The book by Churchhouse also gives an overview of a number of historical ciphers.

R. Churchhouse. *Codes and Ciphers. Julius Caesar, the Enigma and the Internet.* Cambridge University Press, 2001.

D. Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet.* Scribner, 1996.

S. Singh. *The Codebook: The Evolution of Secrecy from Mary, Queen of Scots to Quantum Cryptography.* Doubleday, 2000.

CHAPTER 4

The Enigma Machine

Chapter Goals

- To explain the working of the Enigma machine.
- To explain how the Germans used the Enigma machine, in particular how session keys were transmitted from the sender to the reciever.
- To explain how this enabled Polish and later British cryptanalysts to read the German traffic.
- To explain the use of the Bombe in mounting known plaintext attacks.

1. Introduction

With the advent of the 1920s people saw the need for a mechanical encryption device. Taking a substitution cipher and then rotating it became seen as the ideal solution. This idea had actually been used previously in a number of manual ciphers, but using machines it was seen how this could be done more efficiently. The rotors could be implemented using wires and then encryption could be done mechanically using an electrical circuit. By rotating the rotor we obtain a new substitution cipher.

As an example, suppose the rotor used to produce the substitutions is given by

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
TMKGOYDSIPELUAVCRJWXZNHBQF
```

To encrypt the first letter we use the substitutions given by

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
TMKGOYDSIPELUAVCRJWXZNHBQF
```

However, to encrypt the second letter we rotate the rotor by one position and use the substitutions

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
MKGOYDSIPELUAVCRJWXZNHBQFT
```

To encrypt the third letter we use the substitutions

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
KGOYDSIPELUAVCRJWXZNHBQFTM
```

and so on. This gives us a polyalphabetic substitution cipher with 26 alphabets.

The most famous of these machines was the Enigma machine used by the Germans in World War II. We shall describe the most simple version of Enigma which only used three such rotors, chosen from the following set of five.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
EKMFLGDQVZNTOWYHXUSPAIBRCJ
AJDKSIRUXBLHWTMCQGZNPYFVOE
BDFHJLCPRTXVZNYEIWGAKMUSQO
ESOVPZJAYQUIRHXLNFTGKDCMWB
```

VZBRGITYUPSDNHLXAWMJQOFECK

Machines in use towards the end of the war had a larger number of rotors, chosen from a larger set. Note, the order of the rotors in the machine is important, so the number of ways of choosing the rotors is

$$5 \cdot 4 \cdot 3 = 60.$$

Each rotor had an initial starting position, and since there are 26 possible starting positions for each rotor, the total number of possible starting positions is $26^3 = 17\,576$.

The first rotor would step on the second rotor on each full iteration under the control of a ring hitting a notch, likewise the stepping of the third rotor was controlled by the second rotor. Both the rings were movable and their positions again formed part of the key, although only the notch and ring positions for the first two rotors were important. Hence, the number of ring positions was $26^2 = 676$. The second rotor also had a kick associated to it making the cycle length of the three rotors equal to

$$26 \cdot 25 \cdot 26 = 16\,900.$$

The effect of the moving rotors was that a given plaintext letter would encrypt to a different ciphertext letter on each press of the keyboard.

Finally, a plug board was used to swap letters twice in each encryption and decryption operation. This increased the complexity and gave another possible 10^{14} keys.

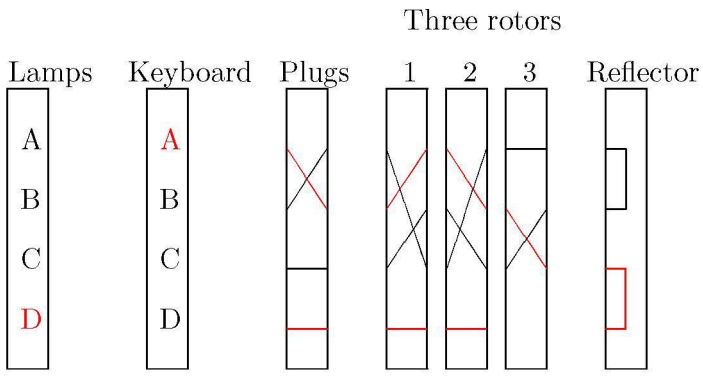
The rotors used, their order, their starting positions, the ring positions and the plug board settings all made up the secret key. Hence, the total number of keys was then around 2^{75}.

To make sure encryption and decryption were the same operation a reflector was used. This was a fixed public substitution given by

ABCDEFGHIJKLMNOPQRSTUVWXYZ
YRUHQSLDPXNGOKMIEBFZCWVJAT

The operation of a simplified Enigma machine is described in Fig. 1. By tracing the red lines one can see how the plaintext character A encrypts to the ciphertext character D. Notice that encryption and decryption can be performed by the machine being in the same positions. Now assume that rotor one moves on one step, so A now maps to D under rotor one, B to A, C to C and D to B. You should work out what happens with the example when we encrypt A again.

FIGURE 1. Simplified Enigma machine



In the rest of this chapter we present more details on the Enigma machine and some of the attacks which can be performed on it. However before presenting the machine itself we need to fix some notation which will be used throughout this chapter. In particular lower case letters will denote variables, upper case letters will denote "letters" (of the plaintext/ciphertext languages) and greek letters will denote permutations in S_{26} which we shall represent as permutations on the

upper case letters. Hence x can equal X and Y, but X can only ever represent X, whereas χ could represent (XY) or (ABC).

Permutations will usually be given in cycle notation. One always has to make a choice as to whether we multiply permutations from left to right, or right to left. We decide to use the left to right method, hence

$$(ABCD)(BE)(CD) = (AEBD).$$

Permutations hence act on the right of letters, something we will denote by x^σ, e.g.

$$A^{(ABCD)(XY)} = B.$$

This is consistent with the usual notation of right action for groups acting on sets. See Appendix A for more details about permutations.

We now collect some basic facts and theorems about permutations which we will need in the sequel.

THEOREM 4.1. *Two permutations σ and τ which are conjugate, i.e. for which $\sigma = \lambda \cdot \tau \cdot \lambda^{-1}$ for some permutation λ, have the same cycle structure.*

We define the support of a permutation to be the set of letters which are not fixed by the permutation. Hence, if σ acts on the set of letters $\mathcal{L}$, then as usual we denote by $\mathcal{L}^\sigma$ the set of fixed points and hence the support is given by

$$\mathcal{L} \setminus \mathcal{L}^\sigma.$$

THEOREM 4.2. *If two permutations, with the same support, consist only of disjoint transpositions then their product contains an even number of disjoint cycles of the same lengths.*

If a permutation with support an even number of symbols has an even number of disjoint cycles of the same lengths, then the permutation can be written as a product of two permutations each of which consists of disjoint transpositions.

In many places we need an algorithm to solve the following problem: Given $\alpha_i, \beta_i \in S_{26}$, for $i = 1, \ldots, m$ find $\gamma \in S_{26}$ such that

$$\alpha_i = \gamma^{-1} \cdot \beta_i \cdot \gamma \text{ for } i = 1 \ldots, m.$$

Note, there could be many such solutions γ, but in the situations we will apply it we expect there to be only a few.

For example suppose we have one such equation with

$$\begin{aligned} \alpha_1 &= (AFCNE)(BWXHUJOG)(DVIQZ)(KLMYTRPS), \\ \beta_1 &= (AEYSXWUJ)(BFZNO)(CDPKQ)(GHIVLMRT) \end{aligned}$$

We need to determine the structure of the permutation γ such that

$$\alpha_1 = \gamma^{-1} \cdot \beta_1 \cdot \gamma.$$

We first look at what should A map to under γ. If $A^\gamma = B$, then from α_1 and β_1 we must have $E^\gamma = W$, which in turn implies $Y^\gamma = X$. Carrying on in this way via a pruned depth first search we can determine a set of possible values for γ. Such an algorithm is relatively simple to write down in C, using a recursive procedure call. It however of course been a bit of a pain to do this by hand, as one would need to in the 1930's and 1940's.

2. An Equation For The Enigma

To aid our discussion in later sections we now describe the Enigma machine as a permutation equation. We first assume a canonical map between letters and the integers $\{0, 1, \ldots, 25\}$ such that $0 \leftarrow A$, $1 \leftarrow B$, etc and we assume a standard three wheel Enigma machine.

The wheel which turns the fastest we shall call rotor one, whilst the one which turns the slowest we shall call rotor three. This means that when looking at a real machine rotor three is the left

most rotor and rotor one is the right most rotor. This can cause confusion (especially when reading day/message settings), so please keep this in mind.

The basic permutations which make up the Enigma machine are as follows:

2.1. Choice of Rotors. We assume that the three rotors are chosen from the following set of five rotors. We present these rotors in cycle notation, but they are the commonly labelled rotors I, II, III, IV and V used in the actual Enigma machines, which were given earlier. Each rotor also has a different notch position which controls how the stepping of one rotor drives the stepping of the others.

Rotor	Permutation Representation	Notch Position
I	$(AELTPHQXRU)(BKNW)(CMOY)(DFG)(IV)(JZ)$	$16 \leftarrow Q$
II	$(BJ)(CDKLHUP)(ESZ)(FIXVYOMW)(GR)(NT)$	$4 \leftarrow E$
III	$(ABDHPEJT)(CFLVMZOYQIRWUKXSG)$	$21 \leftarrow V$
IV	$(AEPLIYWCOXMRFZBSTGJQNH)(DV)(KU)$	$9 \leftarrow J$
V	$(AVOLDRWFIUQ)(BZKSMNHYC)(EGTJPX)$	$25 \leftarrow Z$

2.2. Reflector. There were a number of reflectors used in actual Enigma machines. In our description we shall use the reflector given earlier, which is often referred to as called "Reflector B". This reflector has representation via disjoint cycles as

$$\varrho = (AY)(BR)(CU)(DH)(EQ)(FS)(GL)(IP)(JX)(KN)(MO)(TZ)(VW).$$

2.3. An Enigma Key. An Enigma key consists of the following information:

- A choice of rotors ρ_1, ρ_2, ρ_3 from the above choice of five possible rotors. Note, this choice of rotors affects the three notch positions, which we shall denote by n_1, n_2 and n_3. Also, as noted above, the rotor ρ_3 is placed in the left of the actual machine, whilst rotor ρ_1 is placed on the right. Hence, if in a German code book it says use rotors
$$I, II, III,$$
this means in our notation that ρ_1 is selected to be rotor III, that ρ_2 is selected to be rotor II and ρ_3 is selected to be rotor I.
- One must also select the ring positions, which we shall denote by r_1, r_2 and r_3. In the actual machine these are letters, but we shall use our canonical numbering to represent these as integers in $\{0, 1, \ldots, 25\}$.
- The plugboard is simply a product of disjoint transpositions which we shall denote by the permutation τ. In what follows we shall denote a plug linking letter A with letter B by $A \leftrightarrow B$.
- The starting rotor positions we shall denote by p_1, p_2 and p_3. These are the letters which can be seen through the windows on the top of the Enigma machine. Remember our numbering system is that the window on the left corresponds to p_3 and that on the right corresponds to p_1.

2.4. The Encryption Operation. We let σ denote the shift-up permutation given by

$$\sigma = (ABCDEFGHIJKLMNOPQRSTUVWXYZ).$$

The stepping of the second and third rotor is probably the hardest part to grasp when first looking at an Enigma machine, however this has a relatively simple description when one looks at it in a mathematical manner.

Given the above description of the key we wish to deduce the permutation ϵ_j, which represents the encryption of the jth letter, for $j = 0, 1, 2, \ldots,$.

We first set

$$\begin{aligned} m_1 &= n_1 - p_1 - 1 \pmod{26}, \\ m &= n_2 - p_2 - 1 \pmod{26}, \\ m_2 &= m_1 + 1 + 26m. \end{aligned}$$

The values of m_1 and m_2 control the stepping of the second and the third rotors.

We let $\lfloor x \rfloor$ denote the round towards zero function, i.e. $\lfloor 1.9 \rfloor = 1$ and $\lfloor -1.9 \rfloor = -1$. We now set, for encrypting letter j,

$$\begin{aligned} k_1 &= \lfloor (j - m_1 + 26)/26 \rfloor, \\ k_2 &= \lfloor (j - m_2 + 650)/650 \rfloor, \\ i_1 &= p_1 - r_1 + 1, \\ i_2 &= p_2 - r_2 + k_1 + k_2, \\ i_3 &= p_3 - r_3 + k_2. \end{aligned}$$

Notice, how i_3 is stepped on every $650 = 26 \cdot 25$ iterations whilst i_2 is stepped on every 26 iterations and also stepped on an extra notch every 650 iterations.

We can now present ϵ_j as

$$\begin{aligned} \epsilon_j = \tau \cdot (\sigma^{i_1+j}\rho_1\sigma^{-i_1-j}) \cdot (\sigma^{i_2}\rho_2\sigma^{-i_2}) \cdot (\sigma^{i_3}\rho_3\sigma^{-i_3}) \cdot \varrho \cdot \\ \cdot (\sigma^{i_3}{\rho_3}^{-1}\sigma^{-i_3}) \cdot (\sigma^{i_2}{\rho_2}^{-1}\sigma^{-i_2}) \cdot (\sigma^{i_1+j}{\rho_1}^{-1}\sigma^{-i_1-j}) \cdot \tau. \end{aligned}$$

Note that the same equation/machine is used to encrypt the jth letter as is used to decrypt the jth letter. Hence we have

$${\epsilon_j}^{-1} = \epsilon_j.$$

Also note that each ϵ_j consists of a product of disjoint transpositions. We shall always use γ_j to represent the internal rotor part of the Enigma machine, hence

$$\epsilon_j = \tau \cdot \gamma_j \cdot \tau.$$

3. Determining The Plugboard Given The Rotor Settings

For the moment assume that we know values for the rotor order, ring settings and rotor positions. We would like to determine the plugboard settings, we are therefore given γ_j. The goal is therefore to determine τ given some information about ϵ_j for some values of j.

One often sees written that determining the plugboard given the rotor settings is equivalent to solving a substitution cipher. This is true, but often the method given in some sources is too simplistic.

If we let m denote the actual message being encrypted and c the corresponding ciphertext, and m' the ciphertext decrypted under the cipher with no plugboard, i.e. with an obvious notation,

$$\begin{aligned} m &= c^{\epsilon}, \\ m' &= c^{\gamma}. \end{aligned}$$

The following is an example value of m' for a plugboard containing only one plug

```
ZNCT UPZN A EIME, THEKE WAS A GILL CALLED SNZW WHFTE.
```

I have left the spacing's in the English words. You may then deduce that Z should really be O, or T should really be E, or E should really be T, or maybe K should map to R or L to R or F to I. But which should be the correct plug ? The actual correct plug is setting is that O should map to Z, the other mappings are the result of this single plug setting.

We now present some ways of obtaining information about the plugboard given various scenarios.

3.1. Ciphertext Only Attack. In a ciphertext only attack one can proceed as one would for a normal substitution cipher. We need a method to be able to distinguish something which could be natural language from something which is completely random. The best statistic seems to be to use one called the Sinkov statistic. Let f_i, for $i = A, \ldots, Z$, denote the frequencies of the various letters in standard English. For a given piece of text we let n_i, for $i = A, \ldots, Z$, denote the frequencies of the various letters within the sample piece of text. The Sinkov statistic for the sample text is given by

$$s = \sum_{i=A}^{Z} n_i f_i.$$

A high value of this statistic corresponds to something which is more likely to be from a natural language.

To mount a ciphertext only attack we let γ_j denote our current approximation for ϵ_j (initially γ_j has no plug settings, but this will change as the method progresses). We now go through all possible single plug settings, $\alpha^{()}$. There are $26 \cdot 25/2 = 325$ of these. We then decrypt the ciphertext c using the cipher

$$\alpha^{(k)} \cdot \gamma_j \cdot \alpha^{(k)}.$$

This results in 325 possible plaintext messages $m^{(k)}$ for $k = 1, \ldots, 325$. For each one of these we compute the Sinkov statistic s_k, we keep the value of $\alpha^{(k)}$ which results in s_k being minimized. We then set our new γ_j to be $\alpha^{(k)} \cdot \gamma_j \cdot \alpha^{(k)}$ and repeat, until no further improvement can be made in the test statistic.

This methodology seems very good at finding the missing plugboard settings. For example consider an Enigma machine with the rotors set to be Suppose we are given that the day setting is

Rotors	Rings	Pos	Plugboard
III, II, I	PPD	MDL	Unknown

The actual hidden plugboard is given by $A \leftrightarrow B$, $C \leftrightarrow D$, $E \leftrightarrow F$, $G \leftrightarrow H$, $I \leftrightarrow J$ and $K \leftrightarrow L$. We obtain the ciphertext

```
HUCDODANDHOMYXUMGLREDSQQJDNJAEXUKAZOYGBYLEWFNWIBWILSMAETFFBVPR
GBYUDNAAIEVZZKCUFNIUTOKNKAWUTUWQJYAUHMFWJNIQHAYNAGTDGTCTNYKTCU
FGYQBSRRUWZKZFWKPGVLUHYWZCZSOYJNXHOSKVPHGSGSXEOQWOZYBXQMKQDDXM
BJUPSQODJNIYEPUCEXFRHDQDAQDTFKPSZEMASWGKVOXUCEYWBKFQCYZBOGSFES
OELKDUTDEUQZKMUIZOGVTWKUVBHLVXMIXKQGUMMQHDLKFTKRXCUNUPPFKWUFCU
PTDMJBMMTPIZIXINRUIEMKDYQFMIQAEVLWJRCYJCUKUFYPSLQUEZFBAGSJHVOB
CHAKHGHZAVJZWOLWLBKNTHVDEBULROARWOQGZLRIQBVVSNKRNUCIKSZUCXEYBD
QKCVMGLRGFTBGHUPDUHXIHLQKLEMIZKHDEPTDCIPF
```

The plugboard settings are found in the following order $I \leftrightarrow J$, $E \leftrightarrow F$, $A \leftrightarrow B$, $G \leftrightarrow H$, $K \leftrightarrow L$ and $C \leftrightarrow D$. The plaintext is determined to be.

```
ITWASTHEBESTOFTIMESITWASTHEWORSTOFTIMESITWASTHEAGEOFWISDOMITWA
STHEAGEOFFOOLISHNESSITWASTHEEPOCHOFBELIEFITWASTHEEPOCHOFINCRED
ULITYITWASTHESEASONOFLIGHTITWASTHESEASONOFDARKNESSITWASTHESPRI
NGOFHOPEITWASTHEWINTEROFDESPAIRWEHADEVERYTHINGBEFOREUSWEHADNOT
HINGBEFOREUSWEWEREALLGOINGDIRECTTOHEAVENWEWEREALLGOINGDIRECTTH
EOTHERWAYINSHORTTHEPERIODWASSOFARLIKETHEPRESENTPERIODTHATSOMEO
FITSNOISIESTAUTHORITIESINSISTEDONITSBEINGRECEIVEDFORGOODORFORE
VILINTHESUPERLATIVEDEGREEOFCOMPARISONONLY
```

3.2. Known Plaintext Attack. When one knows the plaintext there are two methods one can employ. The first method is simply based on a depth first search technique, whilst the second makes use of some properties of the encryption operation.

3.2.1. *Technique One:* In the first technique we take each wrong letter in turn, from our current approximation γ_j to ϵ_j. In the above example, of the encryption of "A Tale of Two Cities", we have that the first ciphertext letter H should map to the plaintext letter I. This implies that the plugboard must contain plug settings $H \leftrightarrow p_H$ and $I \leftrightarrow p_I$, for letters p_H and p_I with

$$p_H{}^{\gamma_0} = p_I.$$

We in a similar manner deduce the following other equations

$$\begin{array}{lll} p_U{}^{\gamma_1} = p_T, & p_C{}^{\gamma_2} = p_W, & p_D{}^{\gamma_3} = p_A, \\ p_O{}^{\gamma_4} = p_S, & p_D{}^{\gamma_5} = p_T, & p_A{}^{\gamma_6} = p_H, \\ p_N{}^{\gamma_7} = p_E, & p_D{}^{\gamma_8} = p_B, & p_H{}^{\gamma_9} = p_E. \end{array}$$

The various permutations which represent the first few γ_j's for the given rotor and ring positions are as follows:

$$\begin{aligned} \gamma_0 &= (AW)(BH)(CZ)(DE)(FT)(GJ)(IN)(KL)(MQ)(OV)(PU)(RS)(XY), \\ \gamma_1 &= (AZ)(BL)(CE)(DH)(FK)(GJ)(IS)(MX)(NQ)(OY)(PR)(TU)(VW), \\ \gamma_2 &= (AZ)(BJ)(CV)(DW)(EP)(FX)(GO)(HS)(IY)(KL)(MN)(QT)(RU), \\ \gamma_3 &= (AF)(BC)(DY)(EO)(GU)(HK)(IV)(JR)(LX)(MN)(PW)(QS)(TZ), \\ \gamma_4 &= (AJ)(BD)(CF)(EL)(GN)(HX)(IM)(KQ)(OS)(PV)(RT)(UY)(WZ), \\ \gamma_5 &= (AW)(BZ)(CT)(DI)(EH)(FV)(GU)(JO)(KP)(LN)(MX)(QY)(RS), \\ \gamma_6 &= (AL)(BG)(CO)(DV)(EN)(FS)(HY)(IZ)(JT)(KW)(MP)(QR)(UX), \\ \gamma_7 &= (AI)(BL)(CT)(DE)(FN)(GH)(JY)(KZ)(MO)(PS)(QX)(RU)(VW), \\ \gamma_8 &= (AC)(BH)(DU)(EM)(FQ)(GV)(IO)(JZ)(KS)(LT)(NR)(PX)(WY), \\ \gamma_9 &= (AB)(CM)(DY)(EZ)(FG)(HN)(IR)(JX)(KV)(LW)(OT)(PQ)(SU). \end{aligned}$$

We now proceed as follows: Suppose we know that there are exactly six plugs being used. This means that if we pick a letter at random, say T, then there is a $14/26 = 0.53$ chance that this letter is not plugged to another one. Let us therefore make this assumption for the letter T, in which case $p_T = T$. From the above equations involving γ_1 and γ_5 we then deduce that

$$p_U = U \text{ and } p_D = C.$$

We then use the equations involving γ_3 and γ_8, since we now know p_D, to deduce that

$$p_A = B \text{ and } p_B = A.$$

This latter two checks are consistent so we can assume that our original choice of $p_T = T$ was a good one. From the equations involving γ_6, using $p_A = B$ we deduce that

$$p_H = G.$$

Using this in the equations involving γ_0 and γ_9 we deduce that

$$p_I = J \text{ and } p_E = F.$$

We then find that our five plug settings of $A \leftrightarrow B$, $C \leftrightarrow D$, $E \leftrightarrow F$, $G \leftrightarrow H$ and $I \leftrightarrow J$ allow us to decrypt the first ten letters correctly. To deduce the final plug setting will require a piece of ciphertext, and a corresponding piece of known plaintext, such that either the plaintext or the ciphertext involves either the letter K or the letter L.

This technique can also be used when one knows partial information about the rotor positions. For example many of the following techniques will allow us to deduce the differences $p_i - r_i$, but not the actual values of r_i or p_i. However, by following the above technique, on assuming $r_i =' A'$, we will at some point deduce a contradiction. At this point we know that a rotor turnover has either occurred incorrectly or has not occurred when it should have. Hence, we can at this point

backtrack and deduce the correct turnover. For an example of this technique at work see the latter section on the Bombe.

3.2.2. *Technique Two:* A second method is possible when less than 13 plugs used. In the plaintext obtained under γ_j a number of incorrect letters will appear. Again we let m denote the actual plaintext and m' the plaintext derived with the current (possibly empty) plugboard setting. We suppose there are t plugs left to find.

Suppose we concentrate on each places for which the incorrect plaintext letter A occurs, i.e. all occurances of A in the plaintext m which are wrong. Let x denote the corresponding ciphertext letter, there are two possible cases which can occur

- The letter x should be plugged to an unknown letter. In which case the resulting letter in the message m' will behave randomly (assuming γ_j is acts like a random permutation).
- The letter x does not occur in a plugboard setting. In which case the resulting incorrect plaintext character is the one which should be plugged to A in the actual cipher.

Assuming ciphertext letters are uniformly distributed, the first occurance will occur with probability $t/13$, whilst the alternative will occur with probability $1 - t/13$. This gives the following method to determine which letter A should be connected to. For all letters A in the plaintext m compute the frequency of the corresponding letter in the approximate plaintext m'. The letter which has the highest frequency is highly likely to be the one which should be connect to A on the plugboard. Indeed we expect this letter to occur for a proportion of the letters given by $1 - t/13$, all other letters we expect to occur with a proportion of $t/(13 \cdot 26)$ each.

The one problem with this second technique is that it requires a relatively large amount of known plaintext. Hence, in practice the first technique is more likely to be used.

3.3. Knowledge of ϵ_j for some j's. If we know the value of the permutation ϵ_j for values of $j \in \mathcal{S}$, then we have the following equation

$$\epsilon_j = \tau \cdot \gamma_j \cdot \tau \text{ for } j \in \mathcal{S}.$$

Since $\tau = \tau^{-1}$ this allows us to compute possible values of τ using our previous method for solving this conjugation problem. This might not determine the whole plugboard but it will determine enough for other methods to be used.

3.4. Knowledge of $\epsilon_j \cdot \epsilon_{j+3}$ for some j's. A similar method to the previous one applies in this case, as if we know $\epsilon_j \cdot \epsilon_{j+3}$ for all $j \in \mathcal{S}$ and we know γ_j, then we have the equation

$$(\epsilon_j \cdot \epsilon_{j+3}) = \tau \cdot (\gamma_j \cdot \gamma_{j+3}) \cdot \tau \text{ for } j \in \mathcal{S}.$$

4. Double Encryption Of Message Keys

The polish mathematicians Jerzy Rozycki, Henryk Zygalski and Marian Rejewski were the first to find ways of analysing the Enigma machine. To understand their methods one must first understand how the Germans used the machine. On each day the machine was set up with a key, as above, which was chosen by looking up in a code book. Each subnet would have a different day key.

To encipher a message the sending operator decided on a message key. The message key would be a sequence of three letters, say DHI. The message key needs to be transported to the recipient. Using the day key, the message key would be enciphered twice. The double enciphering is to act as a form of error control. Hence, DHI might be enciphered as $XHJKLM$. Note, that D encrypts to X and then K, this is a property of the Enigma machine.

The receiver would obtain $XHJKLM$ and then decrypt this to obtain DHI. Both operators would then move the wheels around to the positions D, H and I, i.e. they would turn the wheels so that D was in the leftmost window, H in the middle one and I in the rightmost window. Then the actual message would be enciphered.

For this example, in our notation, this would mean that the message key is equal to the day key, except that $p_1 = 8 \leftarrow I$, $p_2 = 7 \leftarrow H$ and $p_3 = 3 \leftarrow D$.

Suppose we intercept a set of messages which have the following headers, consisting of the encryption of the three letter rotor positions, followed by its encryption again, i.e. the first six letters of each message are equal to

UCWBLR	ZSETEY	SLVMQH	SGIMVW	PMRWGV
VNGCTP	OQDPNS	CBRVPV	KSCJEA	GSTGEU
DQLSNL	HXYYHF	GETGSU	EEKLSJ	OSQPEB
WISIIT	TXFEHX	ZAMTAM	VEMCSM	LQPFNI
LOIFMW	JXHUHZ	PYXWFQ	FAYQAF	QJPOUI
EPILWW	DOGSMP	ADSDRT	XLJXQK	BKEAKY
.....	.....	.....	.....	.....
DDESRY	QJCOUA	JEZUSN	MUXROQ	SLPMQI
RRONYG	ZMOTGG	XUOXOG	HIUYIE	KCPJLI
DSESEY	OSPPEI	QCPOLI	HUXYOQ	NYIKFW

If we take the last one of these and look at it in more detail. We know that there are three underlying secret letters, say l_1, l_2 and l_3. We also know that

$$l_1{}^{\epsilon_0} = N,\ l_2{}^{\epsilon_1} = Y,\ l_3{}^{\epsilon_2} = I,$$

and

$$l_1{}^{\epsilon_3} = K,\ l_2{}^{\epsilon_4} = F,\ l_3{}^{\epsilon_5} = W.$$

Hence, given that ${\epsilon_j}^{-1} = \epsilon_j$, we have

$$N^{\epsilon_0\epsilon_3} = {l_1}^{\epsilon_0\epsilon_0\epsilon_3} = {l_1}^{\epsilon_3} = K,\ Y^{\epsilon_1\epsilon_4} = F,\ I^{\epsilon_2\epsilon_5} = W.$$

Continuing in this way we can compute a permutation representation of the three products as follows:

$$\begin{aligned}
\epsilon_0 \cdot \epsilon_3 &= (ADSMRNKJUB)(CV)(ELFQOPWIZT)(HY),\\
\epsilon_1 \cdot \epsilon_4 &= (BPWJUOMGV)(CLQNTDRYF)(ES)(HX),\\
\epsilon_2 \cdot \epsilon_5 &= (AC)(BDSTUEYFXQ)(GPIWRVHZNO)(JK).
\end{aligned}$$

5. Determining The Internal Rotor Wirings

However, life was even more difficult for the Poles as they did not even know the rotor wirings or the reflector values. Hence, they needed to break the machine without even having a description of the actual machine. They did have access to a non-miliatary version of Enigma and deduced the basic structure. In this they were very lucky in that they deduced that the wiring between the plugboard and the right most rotor was in the order of the alphabet. If this were not the case there would have been some hidden permutation which would also have needed to be found.

Luckily the French cryptography Gustave Bertrand obtained from a German spy, Hans-Thilo Schmidt, two months worth of day keys. Thus, for two months of traffic the Poles had access to the day settings. From this information they needed to deduce the internal wirings of the Enigma machine.

Note, in the pre-war days the Germans only used three wheels out of a choice of three, hence the number of days keys is actually reduced by a factor of ten. This is, however, only a slight simplification (at least with modern technology).

Suppose we are given that the day setting is

Rotors	Rings	Pos	Plugboard
III, II, I	TXC	EAZ	$(AMTEBC)$

We do not know what the actual rotors are at present, but we know that the one labelled rotor I will be placed in the rightmost slot (our label one). So we have

$$r_1 = 2,\ r_2 = 23,\ r_3 = 19, p_1 = 25,\ p_2 = 0,\ p_3 = 4.$$

Suppose also that the data from the previous section was obtained as traffic for that day. Hence, we obtain the following three values for the products $\epsilon_j \cdot \epsilon_{j+1}$,

$$\begin{aligned}
\epsilon_0 \cdot \epsilon_3 &= (ADSMRNKJUB)(CV)(ELFQOPWIZT)(HY),\\
\epsilon_1 \cdot \epsilon_4 &= (BPWJUOMGV)(CLQNTDRYF)(ES)(HX),\\
\epsilon_2 \cdot \epsilon_5 &= (AC)(BDSTUEYFXQ)(GPIWRVHZNO)(JK).
\end{aligned}$$

From these we wish to deduce the values of $\epsilon_0, \epsilon_1, \ldots, \epsilon_5$. We will use the fact that ϵ_j is a product of disjoint transpositions and Theorem 4.2 and its proof.

We take the first product and look at it in more detail. We take the sets of two cycles of equal degree and write them above one another, with the bottom one reversed in order, i.e.

$$\begin{array}{cccccccccc|cc}
A & D & S & M & R & N & K & J & U & B & C & V\\
T & Z & I & W & P & O & Q & F & L & E & Y & H
\end{array}$$

We now run through all possible shifts of the bottom rows. Each shift gives us a possible value of ϵ_0 and ϵ_3. The value of ϵ_0 is obtained from reading off the disjoint transpositions from the columns, the value of ϵ_3 is obtained by reading off the transpositions from the "off diagonals". For example with the above orientation we would have

$$\begin{aligned}
\epsilon_0 &= (AT)(DZ)(SI)(MW)(RP)(NO)(KQ)(JF)(UL)(BE)(CY)(VH),\\
\epsilon_3 &= (DT)(SZ)(MI)(RW)(NP)(KO)(JQ)(UF)(BL)(AE)(VY)(CH).
\end{aligned}$$

This still leaves us, in this case, with $20 = 2 \cdot 10$ possible values for ϵ_0 and ϵ_3.

Now, to reduce this number we need to really on stupid operators. Various operators had a tendency to always select the same three letter message key. For example popular choices where QWE (the first letters on the keyboard). One operator used the letters of his girlfriend name, Cillie, hence such "cribs" (or guessed/known plaintexts in todays jargon) became known as "cillies". Note, for our analysis here we only need one Cillie the day when we wish to obtain the internal wiring of rotor I.

In our dummy example, suppose we guess (correctly) that the first message key is indeed QWE. This means that $UCWBLR$ is the encryption of QWE twice, this in turn tells us how to align our cycle of length 10 in the first permutation, as under ϵ_0 the letter Q must encrypt to U.

$$\begin{array}{cccccccccc}
A & D & S & M & R & N & K & J & U & B\\
L & E & T & Z & I & W & P & O & Q & F
\end{array}$$

We can check that this is consistent as we see that Q under ϵ_3 must then encrypt to B. If we guessed one more such cillies we can reduce the number of possibilities for $\epsilon_1, \ldots, \epsilon_6$. Assuming we carry on in this way we will finally deduce that

$$\begin{aligned}
\epsilon_0 &= (AL)(BF)(CH)(DE)(GX)(IR)(JO)(KP)(MZ)(NW)(QU)(ST)(VY),\\
\epsilon_1 &= (AK)(BQ)(CW)(DM)(EH)(FJ)(GT)(IZ)(LP)(NV)(OR)(SX)(UY),\\
\epsilon_2 &= (AJ)(BN)(CK)(DZ)(EW)(FP)(GX)(HS)(IY)(LM)(OQ)(RU)(TV),\\
\epsilon_3 &= (AF)(BQ)(CY)(DL)(ES)(GX)(HV)(IN)(JP)(KW)(MT)(OU)(RZ),\\
\epsilon_4 &= (AK)(BN)(CJ)(DG)(EX)(FU)(HS)(IZ)(LW)(MR)(OY)(PQ)(TV),\\
\epsilon_5 &= (AK)(BO)(CJ)(DN)(ER)(FI)(GQ)(HT)(LM)(PX)(SZ)(UV)(WY).
\end{aligned}$$

We now need to use this information to deduce the value of ρ_1, etc. So for the rest of this section we assume we know ϵ_j for $j = 0, \ldots, 5$, and so we mark it in blue.

Recall that we have,

$$\begin{aligned}\epsilon_j &= \tau\cdot(\sigma^{i_1+j}\rho_1\sigma^{-i_1-j})\cdot(\sigma^{i_2}\rho_2\sigma^{-i_2})\cdot(\sigma^{i_3}\rho_3\sigma^{-i_3})\cdot\varrho\cdot\\ &\quad\cdot(\sigma^{i_3}{\rho_3}^{-1}\sigma^{-i_3})\cdot(\sigma^{i_2}{\rho_2}^{-1}\sigma^{-i_2})\cdot(\sigma^{i_1+j}{\rho_1}^{-1}\sigma^{-i_1-j})\cdot\tau\end{aligned}$$

We now assume that no stepping of the second rotor occurs during the first six encryptions under the day setting. This occurs with quite high probability, namely $20/26 \approx 0.77$. If this assumption turns out to be false we will notice this in our later analysis and it will mean we can deduce something about the (unknown to us at this point) position of the notch on the first rotor.

Given that we know the day settings, so that we know τ and the values of i_1, i_2 and i_3 (since we are assuming $k_1 = k_2 = 0$ for $0 \le j \le 5$), we can write the above equation for $0 \le j \le 5$ as

$$\begin{aligned}\lambda_j &= \sigma^{-i_1-j}\cdot\tau\cdot\epsilon_j\cdot\tau\cdot\sigma^{i_1+j}\\ &= \rho_1\cdot\sigma^{-j}\cdot\gamma\cdot\sigma^{j}\cdot{\rho_1}^{-1}.\end{aligned}$$

Where λ_j is now known and we wish to determine ρ_1 for some fixed but unknown value of γ. The permutation γ is in fact equal to

$$\gamma = (\sigma^{i_2-i_1}\rho_2\sigma^{-i_2})\cdot(\sigma^{i_3}\rho_3\sigma^{-i_3})\cdot\varrho\cdot(\sigma^{i_3}{\rho_3}^{-1}\sigma^{-i_3})\cdot(\sigma^{i_2}{\rho_2}^{-1}\sigma^{i_1-i_2}).$$

In our example we get the following values for λ_j,

$$\begin{aligned}\lambda_0 &= (AD)(BR)(CQ)(EV)(FZ)(GP)(HM)(IN)(JK)(LU)(OS)(TW)(XY),\\ \lambda_1 &= (AV)(BP)(CZ)(DF)(EI)(GS)(HY)(JL)(KO)(MU)(NQ)(RW)(TX),\\ \lambda_2 &= (AL)(BK)(CN)(DZ)(EV)(FP)(GX)(HS)(IY)(JM)(OQ)(RU)(TW),\\ \lambda_3 &= (AS)(BF)(CZ)(DR)(EM)(GN)(HY)(IW)(JO)(KQ)(LX)(PV)(TU),\\ \lambda_4 &= (AQ)(BK)(CT)(DL)(EP)(FI)(GX)(HW)(JU)(MO)(NY)(RS)(VZ),\\ \lambda_5 &= (AS)(BZ)(CV)(DO)(EM)(FR)(GQ)(HK)(IL)(JT)(NP)(UW)(XY).\end{aligned}$$

We now form, for $j = 0, \ldots, 4$,

$$\begin{aligned}\mu_j &= \lambda_j\cdot\lambda_{j+1},\\ &= \rho_1\cdot\sigma^{-j}\cdot\gamma\cdot\sigma^{-1}\cdot\gamma\cdot\sigma^{j+1}\cdot{\rho_1}^{-1},\\ &= \rho_1\cdot\sigma^{-j}\cdot\delta\cdot\sigma^{j}\cdot{\rho_1}^{-1},\end{aligned}$$

where $\delta = \gamma\cdot\sigma^{-1}\cdot\gamma\cdot\sigma$ is unknown.

Eliminating δ via $\delta = \sigma^{j-1}{\rho_1}^{-1}\mu_{j-1}\rho_1\sigma^{-j+1}$ we find the following equations for $j = 1, \ldots, 4$,

$$\begin{aligned}\mu_j &= (\rho_1\cdot\sigma^{-1}\cdot{\rho_1}^{-1})\cdot\mu_{j-1}\cdot(\rho_1\cdot\sigma\cdot{\rho_1}^{-1}),\\ &= \alpha\cdot\mu_{j-1}\cdot\alpha^{-1},\end{aligned}$$

where $\alpha = \rho_1\cdot\sigma^{-1}\cdot{\rho_1}^{-1}$.

Hence, μ_j and μ_{j-1} are conjugate and so by Theorem 4.1 have the same cycle structure. For our example we have

$$\begin{aligned}\mu_0 &= (AFCNE)(BWXHUJOG)(DVIQZ)(KLMYTRPS),\\ \mu_1 &= (AEYSXWUJ)(BFZNO)(CDPKQ)(GHIVLMRT),\\ \mu_2 &= (AXNZRTIH)(BQJEP)(CGLSYWUD)(FVMOK),\\ \mu_3 &= (ARLGYWFK)(BIHNXDSQ)(CVEOU)(JMPZT),\\ \mu_4 &= (AGYPMDIR)(BHUTV)(CJWKZ)(ENXQSFLO).\end{aligned}$$

At this point we can check whether our assumption of no-stepping, i.e. a constant value for the values of i_2 and i_3 is valid. If a step did occur in the second rotor then the above permutations would be unlikely to have the same cycle structure.

We need to determine the structure of the permutation α, this is done by looking at the four equations simultaneously. We note that since σ and α are conjugates, under ρ_1, we know that α has cycle structure of a single cycle of length 26.

In our example we only find one possible solution for α, namely

$$\alpha = (AGYWUJOQNIRLSXHTMKCEBZVPFD).$$

To solve for ρ_1 we need to find a permutation such that

$$\alpha = \rho_1 \cdot \sigma^{-1} \cdot {\rho_1}^{-1}.$$

We find there are 26 such solutions

$$\begin{array}{l}
(AELTPHQXRU)(BKNW)(CMOY)(DFG)(IV)(JZ)\\
(AFHRVJ)(BLU)(CNXSTQYDGEMPIW)(KOZ)\\
(AGFIXTRWDHSUCO)(BMQZLVKPJ)(ENY)\\
(AHTSVLWEOBNZMRXUDIYFJCPKQ)\\
(AIZN)(BOCQ)(DJ)(EPLXVMSWFKRYGHU)\\
(AJEQCRZODKSXWGI)(BPMTUFLYHVN)\\
(AKTVOER)(BQDLZPNCSYI)(FMUGJ)(HW)\\
(AL)(BR)(CTWI)(DMVPOFN)(ESZQ)(GKUHXYJ)\\
(AMWJHYKVQFOGLBS)(CUIDNETXZR)\\
(ANFPQGMX)(BTYLCVRDOHZS)(EUJI)(KW)\\
(AOIFQH)(BUKX)(CWLDPREVS)(GN)(MY)(TZ)\\
(APSDQIGOJKYNHBVT)(CX)(EWMZUL)(FR)\\
(AQJLFSEXDRGPTBWNIHCYOKZVUM)\\
(ARHDSFTCZWOLGQK)(BXEYPUNJM)\\
(ASGRIJNKBYQLHEZXFUOMC)(DT)(PVW)\\
(ATE)(BZYRJONLIKC)(DUPWQM)(FVXGSH)\\
(AUQNMEB)(DVYSILJPXHGTFWRK)\\
(AVZ)(CDWSJQOPYTGURLKE)(FXIM)\\
(AWTHINOQPZBCEDXJRMGV)(FYUSK)\\
(AXKGWUTIORNP)(BDYV)(CFZ)(HJSLM)\\
(AYWVCGXLNQROSMIPBEF)(DZ)(HK)(JT)\\
(AZEGYXMJUVD)(BF)(CHLOTKIQSNRP)\\
(BGZFCIRQTLPD)(EHMKJV)(NSOUWX)\\
(ABHNTMLQUXOVFDCJWYZG)(EISP)\\
(ACKLRSQVGBITNUY)(EJXPF)(HOWZ)\\
(ADEKMNVHPGCLSRTOXQW)(BJY)(IUZ)
\end{array}$$

These are the values of $\rho_1 \cdot \sigma^i$, for $i = 0, \ldots, 25$.

So with one days messages we can determine the value of ρ_1 upto multiplication by a power of σ. The Polish had access to two months such data and so were able to determine similar sets for ρ_2 and ρ_3 (as different rotor orders are used on different days). Note, at this point the Germans did not use a selection of three from five rotors.

If we select three representatives $\hat{\rho_1}$, $\hat{\rho_2}$ and $\hat{\rho_3}$, from the sets of possible rotors, then we have

$$\begin{aligned}
\hat{\rho_1} &= \rho_1 \cdot \sigma^{l_1},\\
\hat{\rho_2} &= \rho_2 \cdot \sigma^{l_2},\\
\hat{\rho_3} &= \rho_3 \cdot \sigma^{l_3}.
\end{aligned}$$

However, we still do not know the value for the reflector ϱ, or the correct values of l_1, l_2 and l_3.

Any one of these versions of $\hat{\rho_i}$ will give a valid Enigma machine (with a different reflector). So it does not matter which choice we make for ρ_i, One can verify this by experiment, but I would

like to see this mathematically. So to conclude we have determined the entire internal workings of the Enigma machine.

6. Determining The Day Settings

Now having determined the internal wirings, given the set of two months of day settings obtained by Bertrand, the next task is to determine the actual key when the day settings are not available. At this stage we assume the Germans are still using the encrypt the message setting twice routine.

The essential trick here is to notice that if we write the cipher as

$$\epsilon_j = \tau \cdot \gamma_j \cdot \tau,$$

then

$$\epsilon_j \cdot \epsilon_{j+3} = \tau \cdot \gamma_j \cdot \gamma_{j+3} \cdot \tau.$$

So $\epsilon_j \cdot \epsilon_{j+3}$ is conjugate to $\gamma_j \cdot \gamma_{j+3}$ and so by Theorem 4.1 they have the same cycle structure. More importantly the cycle structure does not depend on the plug board τ.

Hence, if we can use the cycle structure to determine the rotor settings then we are only left with determining the plugboard settings. If we can determine the rotor settings then we know the values of γ_j, for $j = 1, \ldots, 6$, from the encrypted message keys we can compute ϵ_j for $j = 1, \ldots, 6$ as in the previous section. Hence, determining the plugboard settings is then a question of solving one of our conjugacy problems again, for τ. But this is easier than before as we have that τ must be a product of disjoint transpositions.

We have already discussed how to compute $\epsilon_j \cdot \epsilon_{j+3}$ from the encryption of the message keys. Hence, we simply compute these values and compare their cycle structures with those obtained by running through all possible

$$60 \cdot 26^3 \cdot 26^3 = 18,534,946,560$$

choices for the rotors, positions and ring settings. Note, that when this was done by the Poles in the 1930's there was only a choice of the ordering of three rotors. The extra choice of rotors did not come in till a bit later. Hence, the total choice was 10 times less than this figure.

The above simplifies further if we assume that no stepping of the second and third rotor occurs during the calculation of the first six ciphertext characters. Recall this happens around 77 percent of the time. In such a situation the cycle structure depends only on the rotor order and the difference $p_i - r_i$ between the starting rotor position and the ring setting. Hence, we might as well assume that $r_1 = r_2 = r_3 = 0$ when computing all of the cycle structures. So, for 77 percent of the days our search amongst the cycle structures is then only among

$$60 \cdot 26^3 = 1,054,560 \text{ (resp. } 105,456)$$

possible cycle structures.

After the above procedure we have determined all values of the initial day setting bar p_i and r_i, however we know the differences $p_i - r_i$. We also know for any given message the message key p'_1, p'_2, p'_3. Hence, in breaking the actual message we only require the solution for r_1, r_2, the value for r_3 is irrelevant as the third rotor never moves a fourth rotor. Most German messages started with the same two letter word followed by space (space was encoded by 'X'). Hence, we only need to go through 26^2 different positions to get the correct ring setting. Actually one goes through 26^2 wheel positions with a fixed ring, and use the differences to infer the actual ring settings.

Once, r_i is determined from one message the value of p_i can be determined for the day key and then all messages can be trivially broken. Another variant here, if a suitable piece of known plaintext can be deduced, is to apply the technique from Section 3.2.1 with the obvious modification to deduce the ring settings as well.

7. The Germans Make It Harder

In Sept 1938 the German's altered the way that day and message keys were used. Now a day key consisted of a rotor order, the ring settings and the plugboard. But the rotor positions were not part of the day key. A cipher operator would now choose their own initial rotor positions, say AXE and their own message rotor positions, say GPI. The operator would put their machine in the AXE setting and then encrypt GPI twice as before, to obtain say $POWKNP$. The rotors would then be placed in the GPI position and the message would be encrypted. The message header would be $AXEPOWKNP$.

This procedure makes the analysis of the previous section useless. As each message would now have its own "day" rotor position setting, and so one could not collect data from many messages so as to recover $\epsilon_0 \cdot \epsilon_3$ etc, as in the previous section.

What was needed was a new way of characterising the rotor positions. The way invented by Zygalski was to use so-called "females". In the six letters of the enciphered message key a female is the occurance of the same letter in the same position in the string of three. For example, the header $POWKNP$ contains no females, but the header $POWPNL$ contains one female in position zero, i.e. the repeated values of P, seperated by three positions.

Let us see what is implied by the existence of such females: Firstly suppose we receive $POWPNL$ as above and suppose the unknown first key setting is x. Then we have that, if ϵ_i represents the Enigma setting in the ground setting,

$$x^{\epsilon_0} = x^{\epsilon_3} = P.$$

In other words

$$P^{\epsilon_0 \cdot \epsilon_3} = x^{\epsilon_0 \cdot \epsilon_0 \cdot \epsilon_3} = x^{\epsilon_3} = P.$$

In other words P is a fixed point of the permutation $\epsilon_0 \cdot \epsilon_3$.

Since the number of fixed points is a feature of the cycle structure and the cycle structure is invariant under conjugation, we see that the number of fixed points of $\epsilon_0 \cdot \epsilon_3$ is the same irrespective of the plugboard setting.

The use of such females was made easier by so-called Zygalski sheets. The following precomputation was performed, for each rotor order. An Enigma machine was set up with rings in position AAA and then, for each position A to Z of the third (leftmost rotor) a sheet was created. This sheet was a table of 51 by 51 squares, consisting of the letters of the alphabet repeated twice in each direction minus one row and column. A square was removed if the Enigma machine with first and second rotor with that row/column position had a fixed point in the permutation $\epsilon_0 \cdot \epsilon_3$. So for each rotor order there was a set of 26 sheets.

Note, we are going to use the sheets to compute the day ring setting, but they are computed using different rotor positions but with a fixed ring setting. This is because it is easier with an Enigma machine to actually rotate the rotor positions than the rings, then converting between ring and rotor settings is simple.

In fact, it makes sense to also produce a set of sheets for the permutation $\epsilon_1 \cdot \epsilon_4$ and $\epsilon_2 \cdot \epsilon_5$, as without these the number of keys found by the following method is quite large. Hence, for each rotor order we will have 26×3 perforated sheets. The Poles used the following method when only 3 rotors where used, extending it to 5 rotors is simple but was time consuming at the time.

To see how the sheets are used we now proceed with an example. Suppose a set of message headers are received in one day. From these we keep all those which possesses a female in the part corresponding to the encryption of the message key. For example we obtain the following message headers,

HUXTBPGNP	DYRHFLGFS	XTMRSZRCX	YGZVQWZQH
BILJWWRRW	QYRZXOZJV	SZYJPFBPY	MWIBUMWRM
YXMHCUHHR	FUGWINCIA	BNAXGHFGG	TLCXYUXYC

RELCOYXOF	XNEDLLDHK	MWCQOPQVN	AMQCZQCTR
MIPVRYVCR	MQYVVPVKA	TQNJSSIQS	KHMCKKCIL
LQUXIBFIV	NXRZNYXNV	AMUIXVVFV	UROVRUAWU
DSJVDFVTT	HOMFCSQCM	ZSCTTETBH	SJECXKCFN
UPWMQJMSA	CQJEHOVBO	VELVUOVDC	TXGHFDJFZ
DKQKFEJVE	SHBOGIOQQ	QWMUKBUVG	

Now assuming a given rotor order, say the rightmost rotor is rotor I, the middle one rotor II and the leftmost rotor is III, we remove all those headers which could have had a stepping action of the middle rotor in the first six encryptions. To compute these we take third character of the above message headers, i.e. the position p_1 of the rightmost rotor in the encryption of the message key, and the position of the notch on the rightmost rotor assuming the rightmost rotor is I, i.e. $n_1 = 16 \leftarrow' Q'$. We compute the value of m_1 from the Section 2

$$m_1 = n_1 - p_1 - 1 \pmod{26}.$$

and remove all those for which

$$\lfloor (j - m_1 + 26)/26 \rfloor \neq 0 \text{ for } j = 0, 1, 2, 3, 4, 5.$$

This leaves us with the following message headers

HUXTBPGNP	DYRHFLGFS	YGZVQWZQH	QYRZXOZJV
SZYJPFBPY	MWIBUMWRM	FUGWINCIA	BNAXGHFGG
TLCXYUXYC	XNEDLLDHK	MWCQOPQVN	AMQCZQCTR
MQYVVPVKA	LQUXIBFIV	NXRZNYXNV	AMUIXVVFV
DSJVDFVTT	ZSCTTETBH	SJECXKCFN	UPWMQJMSA
CQJEHOVBO	TXGHFDJFZ	DKQKFEJVE	SHBOGIOQQ

We now consider each of the three sets of females in turn. For ease of discussion we only consider those corresponding to $\epsilon_0 \cdot \epsilon_3$. We therefore only examine those message headers which have the same letter in the fourth and seventh positions, i.e.

QYRZXOZJV	TLCXYUXYC	XNEDLLDHK	MWCQOPQVN
AMQCZQCTR	MQYVVPVKA	DSJVDFVTT	ZSCTTETBH
SJECXKCFN	UPWMQJMSA	SHBOGIOQQ	

We now perform the following operation, for each letter P_3. We take the Zygalski sheet for rotor order III, II, I and permutation $\epsilon_0 \cdot \epsilon_3$ and letter P_3 and we place this down on the Table. We think of this sheet first sheet as corresponding to the ring setting

$$r_3 = Q - Q = A,$$

where the Q comes from the first letter in the first message header. Each row r and column c of the first sheet corresponds to the ring setting

$$\begin{aligned} r_1 &= R - r, \\ r_2 &= Y - c. \end{aligned}$$

We now take repeat the following process for each message header with a first letter which we have not yet met before. We take the first letter of the next message header, in this case T and we take the sheet with label

$$P_3 + T - Q.$$

This sheet then has to be placed on top of the other sheets at a certain offset to the original sheet. This offset is computed by taking the top left most square of the new sheet should be placed on top of the square (r, c) of the first sheet given by

$$\begin{aligned} r &= R - C, \\ c &= Y - L, \end{aligned}$$

i.e. we take the difference between the third (resp. second) letter of the new message header and the third (resp. second) letter of the first message header.

This process is repeated until all of the given message headers are used up. Any square which is now clear on all sheets then gives a possible setting for the rings for that day. The actual setting being read off the first sheet using the correspondence above.

This process will give a relatively large number of possible ring settings for each possible rotor order. However, when we intersect the possible values obtained from considering the females in the 0/3 position, with those in the 1/4 and the 2/5 position we find that the number of possibilities shrinks dramatically. Often this allows us to uniquely determine the rotor order and ring setting for the day.

We determine in our example that the rotor order is given by III, II and I, with ring settings given by $r_1 = A$, $r_2 = B$ and $r_3 = C$.

To determine the plugboard settings for the day we can either use a piece of known plaintext as before. However, if no such text is available we can use the females to help drastically reduce the number of possibilities for the plugboard settings.

8. Known Plaintext Attack And The Bombe's

Turing (among others) wanted a technique to break Enigma which did not really on the way the German's used the system, which could and did change. Turing settled on a known plaintext attack, using what was known at the time as a "crib". A crib was a piece of plaintext which was suspected to lie in the given piece of ciphertext.

The methodology of this technique was to from a given piece of ciphertext and a suspected piece of corresponding plaintext to first deduce a so-called "menu". A menu is simply a graph which represents the various relationships between ciphertext and plaintext letters. Then the menu was used to program a electrical device called a Bombe. A Bombe was a device which enumerated the Enigma wheel positions and, given the data in the menu, deduced the possible settings for the rotor orders, wheel positions and some of the plugboard. Finally, the ring positions and the remaining parts of the plugboard needed to be found.

In the following we present a version of this technique which we have deduced from various sources. We follow a running example through so as to explain the method in more detail.

8.1. From Ciphertext to a Menu.

Suppose we receive the following ciphertext

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
```

and suppose we know, for example because we suspect it to be a shipping forecast, that the ciphertext encrypts at some point the plaintext

```
DOGGERFISHERGERMANBIGHTEAST
```

Now we know that in the Enigma machine that a letter cannot decrypt to itself. This means that there are only a few positions for which the plaintext will align correctly with the ciphertext. Suppose we had the following alignment

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
-DOGGERFISHERGERMANBIGHTEAST---------------
```

then we see that this is impossible since the S in the plaintext `FISHER` cannot correspond to the S in the ciphertext. Continuing in this way we find that there are only six possible alignments of the plaintext fragment with the ciphertext:

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERMANBIGHTEAST---------------
---DOGGERFISHERGERMANBIGHTEAST-------------
-----DOGGERFISHERGERMANBIGHTEAST-----------
--------DOGGERFISHERGERMANBIGHTEAST--------
```

```
----------DOGGERFISHERGERMANBIGHTEAST------
----------------DOGGERFISHERGERMANBIGHTEAST
```

In the following we will focus on the first alignment, i.e. we will assume that the first ciphertext letter H decrypts to D and so on. In practice the correct alignment out of all the possible ones would need to be deduced by skill, judgement and experience. However, in any given day a number of such cribs would be obtained and so only the most likely ones would be accepted for use in the following procedure.

As is usual with all our techniques there is a problem if the middle rotor turns over in the part of the ciphertext which we are considering. Our piece of chosen plaintext is 26 letters long, so we could treat it in two sections each of 13 letters. The advantage of this is that we know the middle rotor will only advance once every 26 turns of the fast rotor. Hence, by selecting two groups of 13 letters we can obtain two possible alignments which we know one of which does not contain a middle rotor movement.

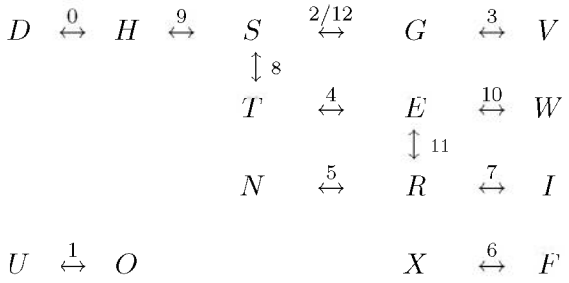
We therefore concentrate on the following two alignments:

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERG-----------------------------
-------------ERMANBIGHTEAS-----------------
```

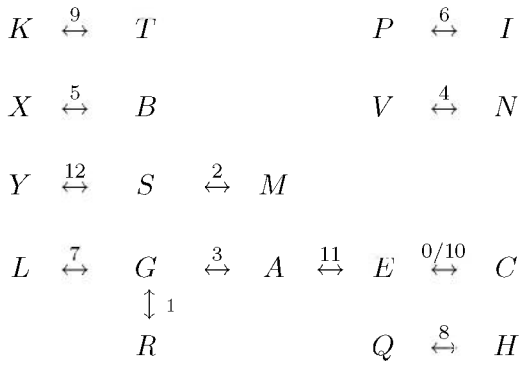
We now deal with each alignment in turn and examine the various pairs of letters. We note that if H encrypts to D in the first position then D will encrypt to H in the same Enigma configuration. We make a record of the letters and the positions for which one letter encrypts to the other. These are placed in a graph with vertices being the letters and edges being labelled by the positions of the related encryptions.

This results in the following two graphs (or menus)

Menu 1:



Menu 2:



These menu's tell us a lot about the configuration of the Enigma machine, in terms of its underlying permutations. Each menu is then used to program a Bombe. In fact we program one

Bombe not only for each menu, but also for each possible rotor order. Thus if five rotor orders are in use, we need to program $2 \cdot 60 = 120$ such Bombe's.

8.2. The Turing/Welchmann Bombe. There are many descriptions of the Bombe as an electrical circuit. In the following we present the basic workings of the Bombe in terms of a modern computer, note however that this is in practice not very efficient. The Bombe's electrical circuit was able to execute the basic operations at the speed of light, (i.e. the time it takes for a current to pass around a circuit), hence simulating this with a modern computer is therefore very inefficient.

I have found the best way to think of the Bombe is as a computer with 26 registers each of which are 26 bits in length. In a single "step" of the Bombe a single bit in this register bank is set. Say we set bit F of register H, this corresponds to us wishing to test whether F is plugged to H in the actual Enigma configuration. The Bombe then passes through a series of states until it stabilises, in the actual Bombe this occurs at the speed of light, in a modern computer simulation this needs to be actually programmed and so occurs at the speed of a computer. Once the register bank stabilises, each set bit is means that if the tested condition is true then so must this condition be true, i.e. if bit J of register K is set then J should be plugged to K in the Enigma machine. In other words the Bombe deduces a "Theorem" of the form

$$\text{If } F \to H \text{ Then } K \to J.$$

With this interpretation the diagonal board described in descriptions of the Bombe is then the obvious condition that if K is plugged to J, then J is plugged to K, i.e. if bit J of register K is set, then so must bit K of register J. In the real Bombe this is achieved by use of wires, however in a computer simulation it means that we always set the "transpose" bit when setting any bit in our register bank. Thus, the register bank is symmetric down the leading diagonal. The diagonal board, which was Welchmann's contribution to basic design of Turing, drastically increases the usefulness of the Bombe in breaking arbitrary cribs.

To understand how the menu acts on the set of registers we define the following permutation for $0 \leq i < 26^3$, for a given choice of rotors ρ_1, ρ_2 and ρ_3 We write $i = i_1 + i_2 \cdot 26 + i_3 \cdot 26^2$, and define

$$\begin{aligned}\delta_{i,s} \quad = \quad & (\sigma^{i_1+s+1}\rho_1\sigma^{-i_1-\varepsilon-1}) \cdot (\sigma^{i_2}\rho_2\sigma^{-i_2}) \cdot (\sigma^{i_3}\rho_3\sigma^{-i_3}) \cdot \varrho \cdot \\ & \cdot(\sigma^{i_3}{\rho_3}^{-1}\sigma^{-i_3}) \cdot (\sigma^{i_2}{\rho_2}^{-1}\sigma^{-i_2}) \cdot (\sigma^{i_1+s+1}{\rho_1}^{-1}\sigma^{-i_1-s-1}).\end{aligned}$$

Note, how similar this is to the equation of the Enigma machine. The main difference is that the second and third rotor's cycle through at a different rate (depending only on i). The variable i is used to denote the rotor position which we wish to currently test and the variable s is used to denote the action of the menu, as we shall now describe.

The menu acts on the registers as follows: For each link $x \xrightarrow{s} y$ in the menu we take register x and for each set bit x_z we apply $\delta_{i,s}$ to obtain x_w. Then bit x_w is set in register y (and due to the diagonal board) bit y is set in register x_w. Also we need to apply the link backwords, so for each set bit y_z in register y we apply $\delta_{i,s}$ to obtain y_w. Then bit y_w is set in register x (and due to the diagonal board) bit x is set in register y_w.

We now let l denote the letter which satisfies at least one of the following, and hopefully all three

(1) A letter which occurs more often than any other letter in the menu.
(2) A letter which occurs in more cycles than any other letter.
(3) A letter which occurs in the largest connected component of the graph of the menu.

In the above two menus we have a number to choose from in Menu 1, so we select $l = S$, in Menu 2 we select $l = E$. For each value of i we then perform the following operation

- Unset all bits in the registers.
- Set bit l of register l.

- Keep applying the menu, as above, until the registers no longer change at all.

Hence, the above algorithm is working out the consequences of the letter l being plugged to itself, given the choice of rotors ρ_1, ρ_2 and ρ_3. It is the third line in the above algorithm which operates at the speed of light in the real Bombe, in a modern simulation this takes a lot longer.

After the the registers converge to a steady state we then test them to see if a possible value of i, i.e. a possible value of the rotor positions has been found. We then step i on by one, which in the real Bombe is achieved by rotating the rotors, and repeat. A value of i which corresponds to a valid value of i is called a "Bombe Stop".

To see what is a valid value of i, suppose we have the rotors in the correct positions. If the plugboard hypothesis, that the letter l is plugged to itself, is true then the registers will converge to a state which gives the plugboard settings for the registers in the graph of the menu which are connected to the letter l. If however the plugboard hypothesis is wrong then the registers will converge to a different state, in particular the bit of each register which corresponds to the correct plugboard configuration will never be set. The best we can then expect is that this wrong hypothesis propagates and all registers in the connected component become set with 25 bits, the one remaining unset bit then corresponds to the correct plugboard setting for the letter l. If the rotor position is wrong then it is highly likely that all the bits in the test register l converge to the set position.

To summarize we have the following situation upon convergence of the registers at step i.

- All 26 bits of test register l are set. This implies that the rotors are not in the correct position and we can step on i by one and repeat the whole process.
- One bit of test register l is set, the rest being unset. This is a possible correct configuration for the rotors. If this is indeed the correct configuration then in addition the set bit corresponds to the correct plug setting for register l, and the single bit set in the registers corresponding to the letters connected to l in the menu will give us the plug settings for those letters as well.
- One bit of the test register l is unset, the rest being set. This is also a possible correct configuration for the rotors. If this is indeed the correct configuration then in addition the unset bit corresponds to the correct plug setting for register l, and any single unset set in the registers corresponding to the letters connected to l in the menu will give us the plug settings for those letters as well.
- The number of set bits in register l lies in $[2, \ldots, 24]$. These are relatively rare occurances, and although they could correspond to actual rotor settings they tell us little directly about the plug settings. For "good" menu's we find they are very rare indeed.

A Bombe stop is a position where the machine decides one has a possible correct configuration of the rotors. The number of such stops per rotor order depends on structure of the graph of the menu. Turing determined the expected number of stops for different types of menus. The following table shows the expected number of stops per rotor order for a connected menu (i.e. only one component) with various numbers of letters and cycles.

	Number of Letters								
Cycles	8	9	10	11	12	13	14	15	16
3	2.2	1.1	0.42	0.14	0.04	≈ 0	≈ 0	≈ 0	≈ 0
2	58	28	11	3.8	1.2	0.3	0.06	≈ 0	≈ 0
1	1500	720	280	100	31	7.7	1.6	0.28	0.04
0	40000	19000	7300	2700	820	200	43	7.3	1.0

This gives an upper bound on the number of stops for an unconnected menu in terms of the the size of the largest connected component and the number of cycles within the largest connected component.

Hence, a good menu is not only one which has a large connected component but which also has a number of cycles. Our second example menu is particularly poor in this respect. Note, that a large number of letters in the connected component not only reduces the expected number of Bombe stops but also increases the number of deductions about possible plugboard configurations.

8.3. Bombe Stop to Plugboard. We now need to work out how from a Bombe stop we can either deduce the actual key, or deduce that the stop has occurred simply by chance and does not correspond to a correct configuration. We first sum up how many stops there are in our example above. For each menu we specify, in the following table, the number of Bombe stops which arise and we also specify the number of bits in the test register l which gave rise to the stop.

	Number of Bits Set									
Menu	1	2	3	4	5-20	21	22	23	24	25
1	137	0	0	0	0	0	0	0	9	1551
2	2606	148	9	2	0	2	7	122	2024	29142

Here we can see the effect of the difference in size of the largest connected component. In both menus the largest connected component has a single cycle in it. For the first menu we obtain a total of 1697 stops, or 28.3 stops per rotor order. The connected component has eleven letters in it, so this yield is much better than the yield expected from the above table. This is due to the extra two letter component in the graph of menu one. For menu two we obtain a total of 34062 stops, or 567.7 stops per rotor order. The connected component in the second menu has six letters in it, so although this figure is bad it is in fact better than the maximum expected from the above table, again this is due to the presence of other components in the graph.

With this large number of stops we need a way of automating the further checking. It turns out that this is relatively simple as the state of the registers allow other conditions to be checked automatically. Apparently in more advanced versions of the Bombe the following checks were performed automatically without the Bombe actually stopping

Recall the Bombe stop gives us information about the state of the supposed plugboard. The following are so-called "legal contradictions", which can be eliminated instantly from the above stops.

- If any Bombe register has 26 bits set then this Bombe configuration is impossible.
- If the Bombe registers imply that a letter is plugged to two different letters then this is clearly a contradiction.

Suppose we know that the plugboard has uses a certain number of plugs (in our example this number is ten) then if the registers imply that there are more than this number of plugs then this is also a contradiction.

Applying these conditions mean we are down to only 19750 possible Bombe stops out of the 35759 total stops above. Of these 109 correspond to the first menu and the rest correspond to the second menu.

We clearly cannot cope with all of those corresponding to the second menu so lets suppose that the second rotor does not turn over in the first thirteen characters. This means we now only need to focus on the first menu.

In practice a number of configurations could be eliminated due to operational requirements set by the Germans (e.g. not using the same rotor orders on consecutive days).

8.4. Finding the final part of the key. We will focus on the first two remaining stops. Both of these correspond to rotor order where the rightmost (fastest) rotor is rotor I, the middle one is rotor II and the leftmost rotor is rotor III.

The first remaining stop is at Bombe configuration $i_1 = p_1 - r_1 = Y$, $i_2 = p_2 - r_2 = W$ and $i_3 = p_3 - r_3 = K$. These follow from the following final register state in this configuration, where rows represent registers and columns the bits

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
A00011011100001000111011000
B00011011100001100111111000
C00001111100001000111011100
D11011111111111111111111111
E11111111011111111111111111
F00111111100110000111011110
G11111101111111111111111111
H11111111101111111111111111
I11110111111111111111111111
J00011010100001100111111000
K00011011100001100111111000
L00011111100001100101111000
M00011111100001000011111000
N11111011111111111111111111
O01011011111101001111101111
P00011011100001000111011000
Q00011011100001100111111000
R11111111111101111111111111
S11111111111011111111111111
T11111111111111111111111110
U01011011111111101111010011
V11111111111111011111111111
W11111111111111111111011111
X00111111100001100111011110
Y00011111100001100111111100
Z00011011100001100110111000
```

The test register has 25 bits set, so in this configuration each bit implies that a letter is not plugged to another letter. The plugboard setting is deduced to contain the following plugs

$$C \leftrightarrow D,\ E \leftrightarrow I,\ F \leftrightarrow N,\ H \leftrightarrow J,\ L \leftrightarrow S,$$

$$M \leftrightarrow R,\ O \leftrightarrow V,\ T \leftrightarrow Z,\ U \leftrightarrow W,$$

whilst the letter G is known to be plugged to itself, assuming this is the correct configuration.

So we need to find one other plug and the ring settings. We can assume that $r_3 = 0 = A$ as it plays no part in the actual decryption process. Since we are using the rotor I as the rightmost rotor we know that $n_1 = 16 \leftarrow Q$, which combined with the fact that we are assuming that no stepping occurs in the first thirteen characters implies that p_1 must satisfy

$$j - ((16 - p_1 - 1) \pmod{26}) + 26 \leq 25 \text{ for } j = 0, \ldots, 12.$$

i.e. $p_1 = 0, 1, 2, 16, 17, 18, 19, 20, 21, 22, 23, 24$ or 25.

With the Enigma setting of $p_1 = Y$, $p_2 = W$, $p_3 = K$ and $r_1 = r_2 = r_3 = A$ and the above (incomplete) plugboard we decrypt the fragment of ciphertext and compare the resulting plaintext with the crib.

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DVGGERLISHERGMWBRXZSWNVOMQOQKLKCSQLRRHPVCAG
DOGGERFISHERGERMANBIGHTEAST----------------
```

This is very much like the supposed plaintext. Examine the first incorrect letter, the second one. This cannot be incorrect due to a second rotor turnover, due to our assumption, hence it must be incorrect due to a missing plugboard element. If we let γ_1 denote the current approximation to the

permutation representing the Enigma machine for letter one and τ the missing plugboard setting then we have

$$U^{\gamma_1} = V \text{ and } U^{\tau\cdot\gamma_1\cdot\tau} = O.$$

This implies that τ should contain either a plug involving the letter U or the letter O, but both of these letters are already used in the plugboard output from the Bombe. Hence, this configuration must be incorrect.

The second remaining stop is at Bombe configuration $i_1 = p_1 - r_1 = R$, $i_2 = p_2 - r_2 = D$ and $i_3 = p_3 - r_3 = L$. The plugboard setting is deduced to contain the following plugs

$$D \leftrightarrow Q,\ E \leftrightarrow T,\ F \leftrightarrow N,\ I \leftrightarrow O,\ S \leftrightarrow V,\ W \leftrightarrow X,$$

whilst the letters G, H and R are known to be plugged to themselves, assuming this is the correct configuration. These follow from the following final register state in this configuration,

```
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
A00011011100001000111011000
B00011111100001000111011100
C00011111100001000111011000
D11111111111111110111111111
E11111111111111111110111111
F01111011100000100111111110
G11111101111111111111111111
H11111110111111111111111111
I11111111111111011111111111
J00011011100001000111011100
K00011011100001000111011000
L00011011100001100111111000
M00011011100001100111111000
N11111011111111111111111111
O00011111000111110111111001
P00011011100001100111111000
Q00001011100001000111011100
R11111111111111111011111111
S11111111111111111111101111
T11110111111111111111111111
U00011111100111110111011001
V11111111111111111101111111
W11111111111111111111111011
X01011111110001001111010110
Y00011111100001000111011100
Z00011011100001100111111000
```

So we need to find four other plug settings and the ring settings.

Again we can assume that $r_3 = A$ as it plays no part in the actual decryption process, and again we deduce that p_1 must be one of $0, 1, 2, 16, 17, 18, 19,\ 20, 21, 22, 23, 24$ or 25.

With the Enigma setting of $p_1 = R$, $p_2 = D$, $p_3 = L$ and $r_1 = r_2 = r_3 = A$ and the above (incomplete) plugboard we decrypt the fragment of ciphertext and compare the resulting plaintext with the crib.

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGNRAMNCOXHXZMORIKOEDEYWEFEYMSDQ
DOGGERFISHERGERMANBIGHTEAST----------------
```

We now look at the first incorrect letter, this is in the 14 position. Using the same notation as before, i.e. γ_j for the current approximation and τ for the missing plugs, we see that if this incorrect operation is due to a plug problem rather than a rotor turnover problem then we must have

$$C^{\tau\cdot\gamma_{13}\cdot\tau} = E.$$

Now, E already occurs on the plugboard, via $E \leftrightarrow T$, so τ must include a plug which maps C to the letter x where

$$x^{\gamma_{13}} = E.$$

But we can compute that

$$\gamma_{13} = (AM)(BE)(CN)(DO)(FI)(GS)(HX)(JU)(KP)(LQ)(RV)(TY)(WZ),$$

from which we deduce that $x = B$. So we include the plug $C \leftrightarrow B$ in our new approximation and repeat to obtain the plaintext

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERAMNBOXHXNMORIKOEMEYWEFEYMSDQ
DOGGERFISHERGERMANBIGHTEAST----------------
```

We then see in the 16th position that we either need to step the rotor or there should be a plug which means that S maps to M under the cipher. We have, for our new γ_{15} that

$$\gamma_{15} = (AS)(BJ)(CY)(DK)(EX)(FW)(GI)(HU)(LM)(NQ)(OP)(RV)(TZ).$$

The letter S already occurs in a plug, so we must have that A is plugged to M. We add this plug into our configuration and repeat

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERMANBOXHXNAORIKVEAEYWEFEYASDQ
DOGGERFISHERGERMANBIGHTEAST----------------
```

Now the 20th character is incorrect, we need that P should map to I and not O under the cipher in this position. Again assuming this is due to a missing plug we find that

$$\gamma_{19} = (AH)(BM)(CF)(DY)(EV)(GX)(IK)(JR)(LS)(NT)(OP)(QW)(UZ).$$

There is already a plug involving the letter I so we deduce that the missing plug should be $K \leftrightarrow P$. Again we add this new plug into our configuration and repeat to obtain

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERMANBIXHJNAORIPVXAEYWEFEYASDQ
DOGGERFISHERGERMANBIGHTEAST----------------
```

Now the 21st character is wrong as we must have that L should map to G. We know G is plugged to itself, from the Bombe stop configuration and given

$$\gamma_{20} = (AI)(BJ)(CW)(DE)(FK)(GZ)(HU)(LX)(MQ)(NT)(OV)(PY)(RS),$$

we deduce that if this error is due to a plug we must have that L is plugged to Z. We add this final plug into our configuration and find that we obtain

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERMANBIGHJNAORIPVXAEYWEFEYAQDQ
DOGGERFISHERGERMANBIGHTEAST----------------
```

All the additional plugs we have added have been on the assumption that no rotor turnover has yet occurred. Any further errors must be due to rotor turnover, as we now have a full set of plugs (as we know our configuration only has ten plugs in use). If when correcting the rotor turnover we still do not decrypt correctly we need to backup and repeat the process.

We see that the next error occurs in position 23. This means that a rotor turnover must have occurred just before this letter was encrypted, in other words we have

$$22 - ((16 - p_1 - 1) \pmod{26}) + 26 = 26.$$

This implies that $p_1 = 19$, i.e. $p_1 = T$, which implies that $r_1 = C$. We now try to decrypt again, and we obtain

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERMANBIGHTZWORIPVXAEYWEFEYAQDQ
DOGGERFISHERGERMANBIGHTEAST----------------
```

But we still do not have correct plaintext. The only thing which could have happened is that we have had an incorrect third rotor movement. Rotor II has its notch in position $n_2 = 4 \leftarrow E$. If the third rotor moved on at position 24 then we have, in our earlier notation

$$\begin{aligned} m_1 &= n_1 - p_1 - 1 \pmod{26} = 16 - 19 - 1 \pmod{26} = 22, \\ m &= n_2 - p_2 - 1 \pmod{26} = 4 - p_2 - 1 \pmod{26}, \\ m_2 &= m_1 + 1 + 26 \cdot m = 23 + 26 \cdot m \\ 650 &= 23 - m_2 + 650 \end{aligned}$$

This last equation implies that $m_2 = 23$, which implies that $m = 0$, which itself implies that $p_2 = 3$, i.e. $p_2 = D$. But this is exactly the setting we have for the second rotor. So the problem is not that the third rotor advances, it is that it should not have advanced. We therefore need to change this to say $p_2 = E$ and $r_2 = B$, (although this is probably incorrect it will help us to decrypt the fragment). We find that we then obtain

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERMANBIGHTEASTFORCEFIVEFALLING
DOGGERFISHERGERMANBIGHTEAST----------------
```

Hence, we can conclude, apart from a possible incorrect setting for the second ring we have the correct Enigma setting for this day.

9. Ciphertext Only Attack

The following attack allows one to break the Enigma machine when only a single ciphertext is given. The method relies on the fact that enough ciphertext is given and that a not a full set of plugs is used. Suppose we have a reasonably large amount of ciphertext, say 500 odd characters, and that p plugs are in use. Suppose in addition that we could determine the rotor settings. This would mean that around $((26 - 2p)/26)^2$ of the letters would decrypt exactly, as the letters would neither pass through a plug either before or after the rotor stage. Hence, one could distinguish the correct rotor positions by using some statistic to distinguish a random plaintext from a plaintext in which $((26 - 2p)/26)^2$ of the letters are correct.

Gillogly suggests using the index of Coincidence. To use this statistic we compute the frequency f_i of each letter in the resulting plaintext of length n and compute

$$IC = \sum_{i=A}^{Z} \frac{f_i(f_i - 1)}{n(n-1)}.$$

To use this approach we set the rings to position A, A, A and then run through all possible rotor orders and rotor starting positions. For each setting we compute the resulting plaintext and the associated value of IC. We keep those settings which have a high value of IC.

Gillogly then suggests for the settings which give a high value of IC to run through the associated ring settings, adjusting the starting positions as necessary, with a similar test. The problem with this approach is that it is susceptible to the effect of turnover of the various rotors. Either a rotor could turn over when we did not expect it, or it could have turned over by error. This is similar to the situation we obtained in our example using the Bombe in a known plaintext attack.

Consider the following ciphtext, of 734 characters in length

```
RSDZANDHWQJPPKOKYANQIGTAHIKPDFHSAWXDPSXXZMMAUEVYYRLWVFFTSDYQPS
CXBLIVFDQRQDEBRAKIUVVYRVHGXUDNJTRVHKMZXPRDUEKRVYDFHXLNEMKDZEWV
OFKAOXDFDHACTVUOFLCSXAZDORGXMBVXYSJJNCYOHAVQYUVLEYJHKKTYALQOAJ
QWHYVVGLFQPTCDCAZXIZUOECCFYNRHLSTGJILZJZWNNBRBZJEEXAEATKGXMYJU
GHMCJRQUODOYMJCXBRJGRWLYRPQNABSKSVNVFGFOVPJCVTJPNFVWCFUUPTAXSR
VQDATYTTHVAWTQJPXLGBSIDWQNVHXCHEAMVWXKIUSLPXYSJDUQANWCBMZFSXWH
JGNWKIOKLOMNYDARREPGEZKCTZNPQKOMJZSQHYEADZTLUPGBAVCVNJHXQKYILX
LTHZXJKYFQEBDBQOHMXBTVXSRGMPVOGMVTEYOCQEOZUSLZDQZBCXXUXBZMZSWX
OCIWRVGLOEZWVVOQJXSFYKDQDXJZYNPGLWEEVZDOAKQOUOTUEBTCUTPYDHYRUS
AOYAVEBJVWGZHGLHBDHHRIVIAUUBHLSHNNNAZWYCCOFXNWXDLJMEFZRACAGBTG
NDIHOWFUOUHPJAHYZUGVJEYOBGZIOUNLPLNNZHFZDJCYLBKGQEWTQMXJKNYXPC
KAPJGAGKWUCLGTFKYFASCYGTXGZXXACCNRHSXTPYLSJWIEMSABFH
```

We ran through all possible $60 \cdot 26^3$ possible values for the rotors and for the rotor positions, with ring settings equal to A, A, A. We obtain the following "high" values for the IC statistic.

IC	ρ_1	ρ_2	ρ_3	p_1'	p_2'	p_3'
0.04095	I	V	IV	P	R	G
0.0409017	IV	I	II	N	O	R
0.0409017	IV	V	I	M	G	Z
0.0408496	V	IV	II	I	J	B
0.040831	IV	I	V	X	D	A
0.0408087	II	I	V	E	O	J
0.040805	I	IV	III	T	Y	H
0.0407827	V	I	II	J	H	F
0.040779	III	IV	II	R	L	Q
0.0407121	II	III	V	V	C	C
0.0406824	IV	V	III	K	S	D
0.0406675	IV	II	III	H	H	D
0.04066	III	I	IV	P	L	G
0.0406526	IV	V	II	E	E	O
0.0406415	I	II	III	V	D	C
0.0406303	I	II	IV	T	C	G
0.0406266	V	IV	II	I	I	A
0.0406229	II	III	IV	K	Q	I
0.0405969	V	II	III	K	O	R
0.0405931	I	III	V	K	B	O
0.0405931	II	IV	I	K	B	Q
⋮	⋮	⋮	⋮	⋮	⋮	⋮

For the 300 or so such high values we then ran through all possible values for the rings r_1 and r_2 (note the third ring plays no part in the process) and we set the rotor starting positions to be

$$\begin{aligned} p_1 &= p_1' + r_1 + i_1, \\ p_2 &= p_2' + r_2 + i_2, \\ p_3 &= p_3' \end{aligned}$$

The addition of the r_j value is to take into account the change in ring position from A to r_j. The additional value of i_j is taken from the set $\{-1, 0, 1\}$ and is used to accommodate issues to do with rotor turnovers which our crude IC statistic is unable to pick up.

Running through all these possibilities we find the values with the highest values of IC are given by

IC	ρ_1	ρ_2	ρ_3	p_1	p_2	p_3	r_1	r_2	r_3
0.0447751	I	II	III	K	D	C	P	B	A
0.0444963	I	II	III	L	D	C	Q	B	A
0.0444406	I	II	III	J	D	C	O	B	A
0.0443848	I	II	III	K	E	D	P	B	A
0.0443588	I	II	III	K	I	D	P	F	A
0.0443551	I	II	III	K	H	D	P	E	A
0.0443476	I	II	III	K	F	D	P	C	A
0.0442807	I	II	III	L	E	D	Q	B	A
0.0442324	I	II	III	J	H	D	O	E	A
0.0442064	I	II	III	K	G	D	P	D	A
0.0441357	I	II	III	J	G	D	O	D	A
0.0441097	I	II	III	J	E	D	O	B	A
0.0441097	I	II	III	L	F	D	Q	C	A
0.0441023	I	II	III	L	C	C	Q	A	A
0.0440837	I	II	III	J	F	D	O	C	A
0.0440763	I	II	III	J	I	D	O	F	A
0.0440242	I	II	III	K	C	C	P	A	A
0.0439833	I	II	III	L	G	D	Q	D	A
0.0438904	I	II	III	L	I	D	Q	F	A
0.0438607	I	II	III	L	H	D	Q	E	A
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Finally using our previous technique for finding the plugboard settings given the rotor settings in a ciphertext only attack (using the Sinkov statistic) we determine that the actual settings are

ρ_1	ρ_2	ρ_3	p_1	p_2	p_3	r_1	r_2	r_3
I	II	III	L	D	C	Q	B	A

with plugboard given by eight plugs which are

$$A \leftrightarrow B,\ C \leftrightarrow D,\ E \leftrightarrow F,\ G \leftrightarrow H,$$

$$I \leftrightarrow J,\ K \leftrightarrow L,\ M \leftrightarrow N,\ O \leftrightarrow P.$$

With these settings one finds that the plaintext is the first two paragraphs of "A Tale of Two Cities".

Chapter Summary

- We have described the Enigma machine and shown how poor session key agreement was used to break into the German traffic.
- We have also seen how stereotypical messages were also used to attack the system.
- We have seen how the plugboard and the rotors worked independently of each other, which led to attackers being able to break each component seperately.

Further Reading

The paper by Rejweski presents the work of the Polish cryptographers very clearly. The pure ciphertext only attack is presented in the papers by Gillogly and Williams. There are a number of excellent sites on the internet which go into various details, of particular note are Tony Sale's web site and the Bletchley Park site.

J. Gillogly. *Ciphertext-only cryptanalysis of Enigma.* Cryptologia, **14**, 1995.

M. Rejewski. *An application of the theory of permutations in breaking the Enigma cipher.* Applicationes Mathematicae, **16**, 1980.

H. Williams. *Applying statistical language recognition techniques in the ciphertext-only cryptanalysis of Enigma.* Cryptologia, **24**, 2000.

Bletchley Park Web Site. `http://www.bletchleypark.org.uk/`.

Tony Sale's Web Site. `http://www.codesandciphers.org.uk/`.

CHAPTER 5

Information Theoretic Security

Chapter Goals

- To introduce the concept of perfect secrecy.
- To discuss the security of the one-time pad.
- To introduce the concept of entropy.
- To explain the notion of key equivocation, spurious keys and unicity distance.
- To use these tools to understand why the prior historical encryption algorithms are weak.

1. Introduction

Information theory is one of the foundations of computer science. In this chapter we will examine its relationship to cryptography. But we shall not assume any prior familiarity with information theory.

We first need to overview the difference between information theoretic security and computational security. Informally, a cryptographic system is called *computationally secure* if the best *possible* algorithm for breaking it requires N operations, where N is such a large number that it is infeasible to carry out this many operations. With current computing power we assume that 2^{80} operations is an infeasible number of operations to carry out. Hence, a value of N larger than 2^{80} would imply that the system is computationally secure. Notice that no actual system can be proved secure under this definition, since we never know whether there is a better algorithm than the one known. Hence, in practice we say a system is computationally secure if the best *known* algorithm for breaking it requires an unreasonably large amount of computational resources.

Another practical approach, related to computational security, is to reduce breaking the system to solving some well-studied hard problem. For example, we can try to show that a given system is secure if a given integer N cannot be factored. Systems of this form are often called provably secure. However, we only have a proof relative to some hard problem, hence this does not provide an absolute proof.

Essentially, a computationally secure scheme, or one which is provably secure, is only secure when we consider an adversary whose computational resources are bounded. Even if the adversary has large, but limited, resources she still will not break the system.

When considering schemes which are computationally secure we need to be very clear about certain issues:

- We need to be careful about the key sizes etc. If the key size is small then our adversary may have enough computational resources to break the system.
- We need to keep abreast of current algorithmic developments and developments in computer hardware.
- At some point in the future we should expect our system to become broken, either through an improvement in computing power or an algorithmic breakthrough.

It turns out that most schemes in use today are computationally secure, and so every chapter in this book (except this one) will solely be interested in computationally secure systems.

On the other hand, a system is said to be *unconditionally secure* when we place no limit on the computational power of the adversary. In other words a system is unconditionally secure if it cannot be broken even with infinite computing power. Hence, no matter what algorithmic improvements are made or what improvements in computing technology occur, an unconditionally secure scheme will never be broken. Other names for unconditional security you find in the literature are perfect security or information theoretic security.

You have already seen that the following systems are not computationally secure, since we already know how to break them with very limited computing resources:

- shift cipher,
- substitution cipher,
- Vigenère cipher.

Of the systems we shall meet later, the following are computationally secure but are not unconditionally secure:

- DES and Rijndael,
- RSA,
- ElGamal encryption.

However, the one-time pad which we shall meet in this chapter is unconditionally secure, but only if it is used correctly.

2. Probability and Ciphers

Before we can introduce formally the concept of unconditional security we first need to understand in more detail the role of probability in understanding simple ciphers. We make the following definitions:

- Let $\mathbb{P}$ denote the set of possible plaintexts.
- Let $\mathbb{K}$ denote the set of possible keys.
- Let $\mathbb{C}$ denote the set of ciphertexts.

Each of these can be thought of as a probability distribution, where we denote the probabilities by $p(P = m), p(K = k), p(C = c)$.

So for example, if our message space is $\mathbb{P} = \{a, b, c\}$ and the message a occurs with probability $1/4$ then we write

$$p(P = a) = \frac{1}{4}.$$

We make the reasonable assumption that P and K are independent, i.e. the user will not decide to encrypt certain messages under one key and other messages under another. The set of ciphertexts under a specific key k is defined by

$$\mathbb{C}(k) = \{e_k(x) : x \in \mathbb{P}\},$$

where the encryption function is defined by $e_k(m)$. We then have the relationship

$$p(C = c) = \sum_{k:c\in\mathbb{C}(k)} p(K = k) \cdot p(P = d_k(c)), \tag{7}$$

where the decryption function is defined by $d_k(c)$. As an example, which we shall use throughout this section, assume that we have only four messages $\mathbb{P} = \{a, b, c, d\}$ which occur with probability

- $p(P = a) = 1/4$,
- $p(P = b) = 3/10$,
- $p(P = c) = 3/20$,
- $p(P = d) = 3/10$.

Also suppose we have three possible keys given by $\mathbb{K} = \{k_1, k_2, k_3\}$, which occur with probability

- $p(K = k_1) = 1/4$,
- $p(K = k_2) = 1/2$,
- $p(K = k_3) = 1/4$.

Now, suppose we have $\mathbb{C} = \{1, 2, 3, 4\}$, with the encryption function given by the following table

	a	b	c	d
k_1	3	4	2	1
k_2	3	1	4	2
k_3	4	3	1	2

We can then compute, using formula (7),

$$\begin{aligned} p(C = 1) = p(K = k_1)p(P = d) + p(K = k_2)p(P = b) \\ + p(K = k_3)p(P = c) = 0.2625, \end{aligned}$$

$$\begin{aligned} p(C = 2) = p(K = k_1)p(P = c) + p(K = k_2)p(P = d) \\ + p(K = k_3)p(P = d) = 0.2625, \end{aligned}$$

$$\begin{aligned} p(C = 3) = p(K = k_1)p(P = a) + p(K = k_2)p(P = a) \\ + p(K = k_3)p(P = b) = 0.2625, \end{aligned}$$

$$\begin{aligned} p(C = 4) = p(K = k_1)p(P = b) + p(K = k_2)p(P = c) \\ + p(K = k_3)p(P = a) = 0.2125. \end{aligned}$$

Hence, the ciphertexts produced are distributed almost uniformly. For $c \in \mathbb{C}$ and $m \in \mathbb{P}$ we can compute the conditional probability $p(C = c|P = m)$. This is the probability that c is the ciphertext given that m is the plaintext

$$p(C = c|P = m) = \sum_{k:m=d_k(c)} p(K = k).$$

This sum is the sum over all keys k for which the decryption function on input of c will output m. For our prior example we can compute these probabilities as

$$\begin{aligned} &p(C = 1|P = a) = 0, \quad p(C = 2|P = a) = 0, \\ &p(C = 3|P = a) = 0.75, \quad p(C = 4|P = a) = 0.25, \end{aligned}$$

$$\begin{aligned} &p(C = 1|P = b) = 0.5, \quad p(C = 2|P = b) = 0, \\ &p(C = 3|P = b) = 0.25, \quad p(C = 4|P = b) = 0.25, \end{aligned}$$

$$\begin{aligned} &p(C = 1|P = c) = 0.25, \quad p(C = 2|P = c) = 0.25, \\ &p(C = 3|P = c) = 0, \quad p(C = 4|P = c) = 0.5, \end{aligned}$$

$$\begin{aligned} &p(C = 1|P = d) = 0.25, \quad p(C = 2|P = d) = 0.75, \\ &p(C = 3|P = d) = 0, \quad p(C = 4|P = d) = 0. \end{aligned}$$

But, when we try to break a cipher we want the conditional probability the other way around, i.e. we want to know the probability of a given message occurring given only the ciphertext. We can compute the probability of m being the plaintext given c is the ciphertext via,

$$p(P = m|C = c) = \frac{p(P = m)p(C = c|P = m)}{p(C = c)}.$$

This conditional probability can be computed by anyone who knows the encryption function and the probability distributions of K and P. Using these probabilities one may be able to deduce some information about the plaintext once you have seen the ciphertext.

Returning to our previous example we compute

$$\begin{aligned} p(P=a|C=1)&=0, & p(P=b|C=1)&=0.571,\\ p(P=c|C=1)&=0.143, & p(P=d|C=1)&=0.286,\\ \\ p(P=a|C=2)&=0, & p(P=b|C=2)&=0,\\ p(P=c|C=2)&=0.143, & p(P=d|C=2)&=0.857,\\ \\ p(P=a|C=3)&=0.714, & p(P=b|C=3)&=0.286,\\ p(P=c|C=3)&=0, & p(P=d|C=3)&=0,\\ \\ p(P=a|C=4)&=0.294, & p(P=b|C=4)&=0.352,\\ p(P=c|C=4)&=0.352, & p(P=d|C=4)&=0. \end{aligned}$$

Hence

- If we see the ciphertext 1 then we know the message is not equal to a. We also can guess that it is more likely to be b rather than c or d.
- If we see the ciphertext 2 then we know the message is not equal to a or b. We also can be pretty certain that the message is equal to d.
- If we see the ciphertext 3 then we know the message is not equal to c or d and have a good chance that it is equal to a.
- If we see the ciphertext 4 then we know the message is not equal to d, but cannot really guess with certainty as to whether the message is a, b or c.

So in our previous example the ciphertext does reveal a lot of information about the plaintext. But this is exactly what we wish to avoid, we want the ciphertext to give no information about the plaintext.

A system with this property, that the ciphertext reveals nothing about the plaintext, is said to be *perfectly secure*.

DEFINITION 5.1 (Perfect Secrecy). *A cryptosystem has perfect secrecy if*

$$p(P=m|C=c)=p(P=m)$$

for all plaintexts m and all ciphertexts c.

This means the probability that the plaintext is m, given that you know the ciphertext is c, is the same as the probability that it is m without seeing c. In other words knowing c reveals no information about m. Another way of describing perfect secrecy is via:

LEMMA 5.2. *A cryptosystem has perfect secrecy if $p(C=c|P=m)=p(C=c)$ for all m and c.*

PROOF. This trivially follows from the definition

$$p(P=m|C=c)=\frac{p(P=m)p(C=c|P=m)}{p(C=c)}$$

and the fact that perfect secrecy means $p(P=m|C=c)=p(P=m)$. □

The first result about a perfect security is

LEMMA 5.3. *Assume the cryptosystem is perfectly secure, then*

$$\#\mathbb{K}\geq\#\mathbb{C}\geq\#\mathbb{P},$$

where

- $\#\mathbb{K}$ *denotes the size of the set of possible keys,*
- $\#\mathbb{C}$ *denotes the size of the set of possible ciphertexts,*
- $\#\mathbb{P}$ *denotes the size of the set of possible plaintexts.*

PROOF. First note that in any encryption scheme, we must have

$$\#\mathbb{C} \geq \#\mathbb{P}$$

since encryption must be an injective map.

We assume that every ciphertext can occur, i.e. $p(C = c) > 0$ for all $c \in \mathbb{C}$, since if this does not hold then we can alter our definition of $\mathbb{C}$. Then for any message m and any ciphertext c we have

$$p(C = c|P = m) = p(C = c) > 0.$$

This means for each m, that for all c there must be a key k such that

$$e_k(m) = c.$$

Hence, $\#\mathbb{K} \geq \#\mathbb{C}$ as required. □

We now come to the main theorem due to Shannon on perfectly secure ciphers. Shannon's Theorem tells us exactly which encryption schemes are perfectly secure and which are not.

THEOREM 5.4 (Shannon). *Let*

$$(\mathbb{P}, \mathbb{C}, \mathbb{K}, e_k(\cdot), d_k(\cdot))$$

denote a cryptosystem with $\#\mathbb{P} = \#\mathbb{C} = \#\mathbb{K}$. *Then the cryptosystem provides perfect secrecy if and only if*

- *every key is used with equal probability* $1/\#\mathbb{K}$,
- *for each* $m \in \mathbb{P}$ *and* $c \in \mathbb{C}$ *there is a unique key* k *such that* $e_k(m) = c$.

PROOF. Note the statement is *if and only if* hence we need to prove it in both directions. We first prove the *only if* part.

Suppose the system gives perfect secrecy. Then we have already seen for all $m \in \mathbb{P}$ and $c \in \mathbb{C}$ there is a key k such that $e_k(m) = c$. Now, since we have assumed $\#\mathbb{C} = \#\mathbb{K}$ we have

$$\#\{e_k(m) : k \in \mathbb{K}\} = \#\mathbb{K}$$

i.e. there do not exist two keys k_1 and k_2 such that

$$e_{k_1}(m) = e_{k_2}(m) = c.$$

So for all $m \in \mathbb{P}$ and $c \in \mathbb{C}$ there is exactly one $k \in \mathbb{K}$ such that $e_k(m) = c$.

We need to show that every key is used with equal probability, i.e.

$$p(K = k) = 1/\#\mathbb{K} \text{ for all } k \in \mathbb{K}.$$

Let $n = \#\mathbb{K}$ and $\mathbb{P} = \{m_i : 1 \leq i \leq n\}$, fix $c \in \mathbb{C}$ and label the keys $k_1, \ldots, k_n$ such that

$$e_{k_i}(m_i) = c \text{ for } 1 \leq i \leq n.$$

We then have, noting that due to perfect secrecy $p(P = m_i|C = c) = p(P = m_i)$,

$$\begin{aligned} p(P = m_i) &= p(P = m_i|C = c) \\ &= \frac{p(C = c|P = m_i)p(P = m_i)}{p(C = c)} \\ &= \frac{p(K = k_i)p(P = m_i)}{p(C = c)}. \end{aligned}$$

Hence we obtain, for all $1 \leq i \leq n$,

$$p(C = c) = p(K = k_i).$$

This says that the keys are used with equal probability and hence

$$p(K = k) = 1/\#\mathbb{K} \text{ for all } k \in \mathbb{K}.$$

Now we need to prove the result in the other direction. Namely, if

- $\#\mathbb{K} = \#\mathbb{C} = \#\mathbb{P}$,
- every key is used with equal probability $1/\#\mathbb{K}$,
- for each $m \in \mathbb{P}$ and $c \in \mathbb{C}$ there is a unique key k such that $e_k(m) = c$,

then we need to show the system is perfectly secure, i.e.

$$p(P = m|C = c) = p(P = m).$$

We have, since each key is used with equal probability,

$$\begin{aligned} p(C = c) &= \sum_k p(K = k)p(P = d_k(c)) \\ &= \frac{1}{\#\mathbb{K}} \sum_k p(P = d_k(c)). \end{aligned}$$

Also, since for each m and c there is a unique key k with $e_k(m) = c$, we must have

$$\sum_k p(P = d_k(c)) = \sum_m p(P = m) = 1.$$

Hence, $p(C = c) = 1/\#\mathbb{K}$. In addition, if $c = e_k(m)$ then $p(C = c|P = m) = p(K = k) = 1/\#\mathbb{K}$. So using Bayes' Theorem we have

$$\begin{aligned} p(P = m|C = c) &= \frac{p(P = m)p(C = c|P = m)}{p(C = c)} \\ &= \frac{p(P = m)\frac{1}{\#\mathbb{K}}}{\frac{1}{\#\mathbb{K}}} \\ &= p(P = m). \end{aligned}$$

□

We end this section by discussing a couple of systems which have perfect secrecy.

2.1. Modified Shift Cipher. Recall the shift cipher is one in which we 'add' a given letter (the key) onto each letter of the plaintext to obtain the ciphertext. We now modify this cipher by using a different key for each plaintext letter. For example, to encrypt the message HELLO we choose five random keys, say FUIAT. We then add the key onto the plaintext, modulo 26, to obtain the ciphertext MYTLH. Notice, how the plaintext letter L encrypts to different letters in the ciphertext.

When we use the shift cipher with a different random key for each letter, we obtain a perfectly secure system. To see why this is so, consider the situation of encrypting a message of length n. Then the total number of keys, ciphertexts and plaintexts are all equal, namely:

$$\#\mathbb{K} = \#\mathbb{C} = \#\mathbb{P} = 26^n.$$

In addition each key will occur with equal probability:

$$p(K = k) = \frac{1}{26^n},$$

and for each m and c there is a unique k such that $e_k(m) = c$. Hence, by Shannon's Theorem this modified shift cipher is perfectly secure.

2.2. Vernam Cipher. The above modified shift cipher basically uses addition modulo 26. One problem with this is that in a computer, or any electrical device, mod 26 arithmetic is hard, but binary arithmetic is easy. We are particularly interested in the addition operation, which is denoted by $\oplus$ and is equal to the logical exclusive-or, or XOR, operation:

$\oplus$	0	1
0	0	1
1	1	0

In 1917 Gilbert Vernam patented a cipher which used these principles, called the *Vernam cipher* or *one-time pad*. To send a binary string you need a key, which is a binary string as long as the message. To encrypt a message we XOR each bit of the plaintext with each bit of the key to produce the ciphertext.

Each key is only allowed to be used once, hence the term *one-time* pad. This means that key distribution is a pain, a problem which we shall come back to again and again. To see why we cannot get away with using a key twice, consider the following chosen plaintext attack. We assume that Alice always uses the same key k to encrypt a message to Bob. Eve wishes to determine this key and so carries out the following attack:

- Eve generates m and asks Alice to encrypt it.
- Eve obtains $c = m \oplus k$.
- Eve now computes $k = c \oplus m$.

You may object to this attack since it requires Alice to be particularly stupid, in that she encrypts a message for Eve. But in designing our cryptosystems we should try and make systems which are secure even against stupid users.

Another problem with using the same key twice is the following. Suppose Eve can intercept two messages encrypted with the same key

$$c_1 = m_1 \oplus k,$$
$$c_2 = m_2 \oplus k.$$

Eve can now determine some partial information about the pair of messages m_1 and m_2 since she can compute

$$c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k) = m_1 \oplus m_2.$$

Despite the problems associated with key distribution, the one-time pad has been used in the past in military and diplomatic contexts.

3. Entropy

If every message we send requires a key as long as the message, and we never encrypt two messages with the same key, then encryption will not be very useful in everyday applications such as Internet transactions. This is because getting the key from one person to another will be an impossible task. After all one cannot encrypt it since that would require another key. This problem is called the key distribution problem.

To simplify the key distribution problem we need to turn from perfectly secure encryption algorithms to ones which are, hopefully, computationally secure. This is the goal of modern cryptographers, where one aims to build systems such that

- one key can be used many times,
- one small key can encrypt a long message.

Such systems will not be unconditionally secure, by Shannon's Theorem, and so must be at best only computationally secure.

We now need to develop the information theory needed to deal with these computationally secure systems. Again the main results are due to Shannon in the late 1940s. In particular we shall use Shannon's idea of using *entropy* as a way of measuring information.

The word entropy is another name for uncertainty, and the basic tenet of information theory is that uncertainty and information are essentially the same thing. This takes some getting used to, but consider that if you are uncertain what something means then revealing the meaning gives you information. As a cryptographic application suppose you want to determine the information in a ciphertext, in other words you want to know what its true meaning is,

- you are uncertain what the ciphertext means,
- you could guess the plaintext,
- the level of uncertainty you have about the plaintext is the amount of information contained in the ciphertext.

If X is a random variable, the amount of entropy (in bits) associated with X is denoted by $H(X)$, we shall define this quantity formally in a second. First let us look at a simple example to help clarify ideas.

Suppose X is the answer to some question, i.e. *Yes* or *No*. If you know I will always say *Yes* then my answer gives you no information. So the information contained in X should be zero, i.e. $H(X) = 0$. There is no uncertainty about what I will say, hence no information is given by me saying it, hence there is no entropy.

If you have no idea what I will say and I reply *Yes* with equal probability to replying *No* then I am revealing one bit of information. Hence, we should have $H(X) = 1$.

Note that entropy does not depend on the length of the actual message; in the above case we have a message of length at most three letters but the amount of information is at most one bit. We can now define formally the notion of entropy.

DEFINITION 5.5 (Entropy). *Let X be a random variable which takes on a finite set of values x_i, with $1 \leq i \leq n$, and has probability distribution $p_i = p(X = x_i)$. The entropy of X is defined to be*

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i.$$

We make the convention that if $p_i = 0$ then $p_i \log_2 p_i = 0$.

Let us return to our *Yes* or *No* question above and show that this definition of entropy coincides with our intuition. Recall, X is the answer to some question with responses *Yes* or *No*. If you know I will always say *Yes* then

$$p_1 = 1 \text{ and } p_2 = 0.$$

We compute

$$H(X) = -1 \cdot \log_2 1 - 0 \cdot \log_2 0 = 0.$$

Hence, my answer reveals no information to you.

If you have no idea what I will say and I reply *Yes* with equal probability to replying *No* then

$$p_1 = p_2 = 1/2.$$

We now compute

$$H(X) = -\frac{\log_2 \frac{1}{2}}{2} - \frac{\log_2 \frac{1}{2}}{2} = 1.$$

Hence, my answer reveals one bit of information to you.

There are a number of elementary properties of entropy which follow from the definition.

- We always have $H(X) \geq 0$.
- The only way to obtain $H(X) = 0$ is if for some i we have $p_i = 1$ and $p_j = 0$ when $i \neq j$.
- If $p_i = 1/n$ for all i then $H(X) = \log_2 n$.

Another way of looking at entropy is that it measures by how much one can compress the information. If I send a single ASCII character to signal *Yes* or *No*, for example I could simply send Y or N, I am actually sending 8 bits of data, but I am only sending one bit of information. If I wanted to I could compress the data down to 1/8th of its original size. Hence, naively if a message of length n can be compressed to a proportion ϵ of its original size then it contains $\epsilon \cdot n$ bits of information in it.

Let us return to our baby cryptosystem considered in the previous section. Recall we had the probability spaces

$$\mathbb{P} = \{a, b, c, d\},\ \mathbb{K} = \{k_1, k_2, k_3\} \text{ and } \mathbb{C} = \{1, 2, 3, 4\},$$

with the associated probabilities:

- $p(P = a) = 0.25$, $p(P = b) = p(P = d) = 0.3$ and $p(P = c) = 0.15$,
- $p(K = k_1) = p(K = k_3) = 0.25$ and $p(K = k_2) = 0.5$,
- $p(C = 1) = p(C = 2) = p(C = 3) = 0.2625$ and $p(C = 4) = 0.2125$.

We can then calculate the relevant entropies as:

$$\begin{aligned} H(P) &\approx 1.9527, \\ H(K) &\approx 1.5, \\ H(C) &\approx 1.9944. \end{aligned}$$

Hence the ciphertext 'leaks' about two bits of information about the key and plaintext, since that is how much information is contained in a single ciphertext. Later we will calculate how much of this information is about the key and how much about the plaintext.

We wish to derive an upper bound for the entropy of a random variable, to go with our lower bound of $H(X) \geq 0$. To do this we will need the following special case of Jensen's inequality.

THEOREM 5.6 (Jensen's Inequality). *Suppose*

$$\sum_{i=1}^{n} a_i = 1$$

with $a_i > 0$ for $1 \leq i \leq n$. Then, for $x_i > 0$,

$$\sum_{i=1}^{n} a_i \log_2 x_i \leq \log_2 \left(\sum_{i=1}^{n} a_i x_i \right).$$

With equality occurring if and only if $x_1 = x_2 = \ldots = x_n$.

Using this we can now prove the following theorem:

THEOREM 5.7. *If X is a random variable which takes n possible values then*

$$0 \leq H(X) \leq \log_2 n.$$

The lower bound is obtained if one value occurs with probability one, the upper bound is obtained if all values are equally likely.

PROOF. We have already discussed the facts about the lower bound so we will concentrate on the statements about the upper bound. The hypothesis is that X is a random variable with

probability distribution $p_1, \ldots, p_n$, with $p_i > 0$ for all i. One can then deduce the following sequence of inequalities

$$\begin{aligned} H(X) &= -\sum_{i=1}^{n} p_i \log_2 p_i \\ &= \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} \\ &\leq \log_2 \left(\sum_{i=1}^{n} \left(p_i \times \frac{1}{p_i} \right) \right) \text{ by Jensen's inequality} \\ &= \log_2 n. \end{aligned}$$

To obtain equality, we require equality when we apply Jensen's inequality. But this will only occur when $p_i = 1/n$ for all i, in other words all values of X are equally likely. □

The basics of the theory of entropy closely match that of the theory of probability. For example, if X and Y are random variables then we define the joint probability distribution as

$$r_{i,j} = p(X = x_i \text{ and } Y = y_j)$$

for $1 \leq i \leq n$ and $1 \leq j \leq m$.

The joint entropy is then obviously defined as

$$H(X, Y) = -\sum_{i=1}^{n} \sum_{j=1}^{m} r_{i,j} \log_2 r_{i,j}.$$

You should think of the joint entropy $H(X, Y)$ as the total amount of information contained in one observation of $(x, y) \in X \times Y$. We then obtain the inequality

$$H(X, Y) \leq H(X) + H(Y)$$

with equality if and only if X and Y are independent. We leave the proof of this as an exercise.

Just as with probability theory, where one has the linked concepts of joint probability and conditional probability, so the concept of joint entropy is linked to the concept of conditional entropy. This is important to understand, since conditional entropy is the main tool we shall use in understanding non-perfect ciphers in the rest of this chapter.

Let X and Y be two random variables. Recall we defined the conditional probability distribution as

$$p(X = x | Y = y) = \text{ Probability that } X = x \text{ given } Y = y.$$

The entropy of X given an observation of $Y = y$ is then defined in the obvious way by

$$H(X|y) = -\sum_{x} p(X = x | Y = y) \log_2 p(X = x | Y = y).$$

Given this we define the conditional entropy of X given Y as

$$\begin{aligned} H(X|Y) &= \sum_{y} p(Y = y) H(X|y) \\ &= -\sum_{x} \sum_{y} p(Y = y) p(X = x | Y = y) \log_2 p(X = x | Y = y). \end{aligned}$$

This is the amount of uncertainty about X that is left after revealing a value of Y. The conditional and joint entropy are linked by the following formula

$$H(X, Y) = H(Y) + H(X|Y)$$

and we have the following upper bound

$$H(X|Y) \leq H(X)$$

with equality if and only if X and Y are independent. Again we leave the proof of these statements as an exercise.

Now turning to cryptography again, we have some trivial statements relating the entropy of P, K and C.

- $H(P|K,C) = 0$:
  If you know the ciphertext and the key then you know the plaintext. This must hold since otherwise decryption will not work correctly.
- $H(C|P,K) = 0$:
  If you know the plaintext and the key then you know the ciphertext. This holds for all ciphers we have seen so far, and holds for all the symmetric ciphers we shall see in later chapters. However, for modern public key encryption schemes we do not have this last property when they are used correctly.

In addition we have the following identities

$$\begin{aligned} H(K,P,C) &= H(P,K) + H(C|P,K) && \text{as } H(X,Y) = H(Y) + H(X|Y) \\ &= H(P,K) && \text{as } H(C|P,K) = 0 \\ &= H(K) + H(P) && \text{as } K \text{ and } P \text{ are independent} \end{aligned}$$

and

$$\begin{aligned} H(K,P,C) &= H(K,C) + H(P|K,C) && \text{as } H(X,Y) = H(Y) + H(X|Y) \\ &= H(K,C) && \text{as } H(P|K,C) = 0. \end{aligned}$$

Hence, we obtain

$$H(K,C) = H(K) + H(P).$$

This last equality is important since it is related to the conditional entropy $H(K|C)$, which is called the *key equivocation*. The key equivocation is the amount of uncertainty about the key left after one ciphertext is revealed. Recall that our goal is to determine the key given the ciphertext. Putting two of our prior equalities together we find

$$H(K|C) = H(K,C) - H(C) = H(K) + H(P) - H(C). \tag{8}$$

In other words, the uncertainty about the key left after we reveal a ciphertext is equal to the uncertainty in the plaintext and the key minus the uncertainty in the ciphertext. Returning to our previous example, recall we had previously computed

$$\begin{aligned} H(P) &\approx 1.9527, \\ H(K) &\approx 1.5, \\ H(C) &\approx 1.9944. \end{aligned}$$

Hence

$$H(K|C) \approx 1.9527 + 1.5 - 1.9944 \approx 1.4583.$$

So around one and a half bits of information about the key are left to be found, on average, after a single ciphertext is observed. This explains why the system leaks information, and shows that it cannot be secure. After all there are only 1.5 bits of uncertainty about the key to start with, one ciphertext leaves us with 1.4593 bits of uncertainty. Hence, $1.5 - 1.4593 = 0.042$ bits of information about the key are revealed by a single ciphertext.

4. Spurious Keys and Unicity Distance

In our baby example above, information about the key is leaked by an individual ciphertext, since knowing the ciphertext rules out a certain subset of the keys. Of the remaining possible keys, only one is correct. The remaining possible, but incorrect, keys are called the *spurious keys*.

Consider the (unmodified) shift cipher, i.e. where the same key is used for each letter. Suppose the ciphertext is WNAJW, and suppose we know that the plaintext is an English word. The only 'meaningful' plaintexts are RIVER and ARENA, which correspond to the two possible keys F and W. One of these keys is the correct one and one is spurious.

We can now explain why it was easy to break the substitution cipher in terms of a concept called the *unicity distance* of the cipher. We shall explain this relationship in more detail, but we first need to understand the underlying plaintext in more detail. The plaintext in many computer communications can be considered as a random bit string. But often this is not so. Sometimes one is encrypting an image or sometimes one is encrypting plain English text. In our discussion we shall consider the case when the underlying plaintext is taken from English, as in the substitution cipher. Such a language is called a *natural language* to distinguish it from the bitstreams used by computers to communicate.

We first wish to define the entropy (or information) per letter H_L of a natural language such as English. Note, a random string of alphabetic characters would have entropy

$$\log_2 26 \approx 4.70.$$

So we have $H_L \leq 4.70$. If we let P denote the random variable of letters in the English language then we have

$$p(P = a) = 0.082, \ldots, p(P = e) = 0.127, \ldots, p(P = z) = 0.001.$$

We can then compute

$$H_L \leq H(P) \approx 4.14.$$

Hence, instead of 4.7 bits of information per letter, if we only examine the letter frequencies we conclude that English conveys around 4.14 bits of information per letter.

But this is a gross overestimate, since letters are not independent. For example Q is always followed by U and the bigram TH is likely to be very common. One would suspect that a better statistic for the amount of entropy per letter could be obtained by looking at the distribution of bigrams. Hence, we let P^2 denote the random variable of bigrams. If we let $p(P = i, P' = j)$ denote the random variable which is assigned the probability that the bigram 'ij' appears, then we define

$$H(P^2) = -\sum_{i,j} p(P = i, P' = j) \log p(P = i, P' = j).$$

A number of people have computed values of $H(P^2)$ and it is commonly accepted to be given by

$$H(P^2) \approx 7.12.$$

We want the entropy per letter so we compute

$$H_L \leq H(P^2)/2 \approx 3.56.$$

But again this is an overestimate, since we have not taken into account that the most common trigram is *THE*. Hence, we can also look at P^3 and compute $H(P^3)/3$. This will also be an overestimate and so on...

This leads us to the following definition.

DEFINITION 5.8. *The entropy of the natural language L is defined to be*

$$H_L = \lim_{n \longrightarrow \infty} \frac{H(P^n)}{n}.$$

The exact value of H_L is hard to compute exactly but we can approximate it. In fact one has, by experiment, that for English

$$1.0 \leq H_L \leq 1.5.$$

So each letter in English

- requires 5 bits of data to represent it,
- only gives at most 1.5 bits of information.

This shows that English contains a high degree of redundancy. One can see this from the following, which you can still hopefully read (just) even though I have deleted two out of every four letters,

$$\text{On**} \quad \text{up**} \quad \text{a} \quad \text{t**e} \quad \text{t**re} \quad \text{**s} \quad \text{a} \quad \text{**rl} \quad \text{**ll**} \quad \text{Sn**} \quad \text{Wh**e.}$$

The *redundancy* of a language is defined by

$$R_L = 1 - \frac{H_L}{\log_2 \#\mathbb{P}}.$$

If we take $H_L \approx 1.25$ then the redundancy of English is

$$R_L \approx 1 - \frac{1.25}{\log_2 26} = 0.75.$$

So this means that we should be able to compress an English text file of around 10 MB down to 2.5 MB.

We now return to a general cipher and suppose $c \in \mathbb{C}^n$, i.e. c is a ciphertext consisting of n characters. We define $\mathbb{K}(c)$ to be the set of keys which produce a 'meaningful' decryption of c. Then, clearly $\#\mathbb{K}(c) - 1$ is the number of spurious keys given c. The average number of spurious keys is defined to be $\overline{s}_n$, where

$$\begin{aligned}
\overline{s}_n &= \sum_{c \in \mathbb{C}^n} p(C = c) \left(\#\mathbb{K}(c) - 1\right) \\
&= \sum_{c \in \mathbb{C}^n} p(C = c) \#\mathbb{K}(c) - \sum_{c \in \mathbb{C}^n} p(C = c) \\
&= \left(\sum_{c \in \mathbb{C}^n} p(C = c) \#\mathbb{K}(c) \right) - 1.
\end{aligned}$$

Now if n is sufficiently large and $\#\mathbb{P} = \#\mathbb{C}$ we obtain

$$\begin{aligned}
\log_2(\overline{s}_n + 1) &= \log_2 \sum_{c \in \mathbb{C}^n} p(C = c) \#\mathbb{K}(c) \\
&\geq \sum_{c \in \mathbb{C}^n} p(C = c) \log_2 \#\mathbb{K}(c) && \text{Jensen's inequality} \\
&\geq \sum_{c \in \mathbb{C}^n} p(C = c) H(K|c) \\
&= H(K|C^n) && \text{By definition} \\
&= H(K) + H(P^n) - H(C^n) && \text{Equation (8)} \\
&\approx H(K) + nH_L - H(C^n) && \text{If } n \text{ is very large} \\
&= H(K) - H(C^n) \\
&\qquad + n(1 - R_L) \log_2 \#\mathbb{P} && \text{By definition of } R_L \\
&\geq H(K) - n \log_2 \#\mathbb{C} \\
&\qquad + n(1 - R_L) \log_2 \#\mathbb{P} && \text{As } H(C^n) \leq n \log_2 \#\mathbb{C} \\
&= H(K) - nR_L \log_2 \#\mathbb{P} && \text{As } \#\mathbb{P} = \#\mathbb{C}.
\end{aligned}$$

So, if n is sufficiently large and $\#\mathbb{P} = \#\mathbb{C}$ then

$$\overline{s}_n \geq \frac{\#\mathbb{K}}{\#\mathbb{P}^{nR_L}} - 1.$$

As an attacker we would like the number of spurious keys to become zero, and it is clear that as we take longer and longer ciphertexts then the number of spurious keys must go down.

The unicity distance n_0 of a cipher is the value of n for which the expected number of spurious keys becomes zero. In other words this is the average amount of ciphertext needed before an attacker can determine the key, assuming the attacker has infinite computing power. For a perfect cipher we have $n_0 = \infty$, but for other ciphers the value of n_0 can be alarmingly small. We can obtain an estimate of n_0 by setting $\overline{s}_n = 0$ in

$$\overline{s}_n \geq \frac{\#\mathbb{K}}{\#\mathbb{P}^{nR_L}} - 1$$

to obtain

$$n_0 \approx \frac{\log_2 \#\mathbb{K}}{R_L \log_2 \#\mathbb{P}}.$$

In the substitution cipher we have

$$\begin{aligned}
\#\mathbb{P} &= 26, \\
\#\mathbb{K} &= 26! \approx 4 \cdot 10^{26}
\end{aligned}$$

and using our value of $R_L = 0.75$ for English we can approximate the unicity distance as

$$n_0 \approx \frac{88.4}{0.75 \times 4.7} \approx 25.$$

So we require on average only 25 ciphertext characters before we can break the substitution cipher, again assuming infinite computing power. In any case after 25 characters we expect a unique valid decryption.

Now assume we have a modern cipher which encrypts bit strings using keys of bit length l, we have

$$\begin{aligned}
\#\mathbb{P} &= 2, \\
\#\mathbb{K} &= 2^l.
\end{aligned}$$

Again we assume $R_L = 0.75$, which is an underestimate since we now need to encode English into a computer communications media such as ASCII. Then the unicity distance is

$$n_0 \approx \frac{l}{0.75} = \frac{4l}{3}.$$

Now assume instead of transmitting the plain ASCII we compress it first. If we assume a perfect compression algorithm then the plaintext will have no redundancy and so $R_L \approx 0$. In which case the unicity distance is

$$n_0 \approx \frac{l}{0} = \infty.$$

So you may ask if modern ciphers encrypt plaintexts with no redundancy? The answer is no, even if one compresses the data, a modern cipher often adds some redundancy to the plaintext before encryption. The reason is that we have only considered passive attacks, i.e. an attacker has been only allowed to examine ciphertexts and from these ciphertexts the attacker's goal is to determine the key. There are other types of attack called active attacks, in these an attacker is allowed to generate plaintexts or ciphertexts of her choosing and ask the key holder to encrypt or decrypt them, the two variants being called a chosen plaintext attack and a chosen ciphertext attack respectively.

In public key systems that we shall see later, chosen plaintexts attacks cannot be stopped since anyone is allowed to encrypt anything. We would however, like to stop chosen ciphertext attacks. The current wisdom for public key algorithms is to make the cipher add some redundancy to the plaintext before it is encrypted. In that way it is hard for an attacker to produce a ciphertext which has a valid decryption. The philosophy is that it is then hard for an attacker to mount a chosen ciphertext attack, since it will be hard for an attacker to choose a valid ciphertext for a decryption query. We shall discuss this more in later chapters.

Chapter Summary

- A cryptographic system for which knowing the ciphertext reveals no more information than if you did not know the ciphertext is called a perfectly secure system.
- Perfectly secure systems exist, but they require keys as long as the message and a different key to be used with each new encryption. Hence, perfectly secure systems are not very practical.
- Information and uncertainty are essentially the same thing. An attacker really wants, given the ciphertext, to determine some information about the plaintext. The amount of uncertainty in a random variable is measured by its entropy.
- The equation $H(K|C) = H(K)+H(P)-H(C)$ allows us to estimate how much uncertainty remains about the key after one observes a single ciphertext.
- The natural redundancy of English means that a naive cipher does not need to produce a lot of ciphertext before the underlying plaintext can be discovered.

Further Reading

Our discussion of Shannon's theory has closely followed the treatment in the book by Stinson. Another possible source of information is the book by Welsh. A general introduction to information theory, including its application to coding theory is in the book by van der Lubbe.

J.C.A. van der Lubbe. *Information Theory.* Cambridge University Press, 1997.

D. Stinson. *Cryptography: Theory and Practice.* CRC Press, 1995.

D. Welsh. *Codes and Cryptography.* Oxford University Press, 1988.

CHAPTER 6

Historical Stream Ciphers

Chapter Goals

- To introduce the general model for symmetric ciphers.
- To explain the relation between stream ciphers and the Vernam cipher.
- To examine the working and breaking of the Lorenz cipher in detail.

1. Introduction To Symmetric Ciphers

A symmetric cipher works using the following two transformations

$$c = e_k(m),$$
$$m = d_k(c)$$

where

- m is the plaintext,
- e is the encryption function,
- d is the decryption function,
- k is the secret key,
- c is the ciphertext.

It should be noted that it is desirable that both the encryption and decryption functions are public knowledge and that the secrecy of the message, given the ciphertext, depends totally on the secrecy of the secret key, k. Although this well-established principle, called Kerckhoffs' principle, has been known since the mid-1800s many companies still ignore it. There are instances of companies deploying secret proprietary encryption schemes which turn out to be insecure as soon as someone leaks the details of the algorithms. The best schemes will be the ones which have been studied by a lot of people for a very long time and which have been found to remain secure. A scheme which is a commercial secret cannot be studied by anyone outside the company.

The above setup is called a symmetric key system since both parties need access to the secret key. Sometimes symmetric key cryptography is implemented using two keys, one for encryption and one for decryption. However, if this is the case we assume that given the encryption key it is easy to compute the decryption key (and vice versa). Later we shall meet public key cryptography where only one key is kept secret, called the private key, the other key, called the public key is allowed to be published in the clear. In this situation it is assumed to be computationally infeasible for someone to compute the private key given the public key.

Returning to symmetric cryptography, a moment's thought reveals that the number of possible keys must be very large. This is because in designing a cipher we assume the worst case scenario and give the attacker the benefit of

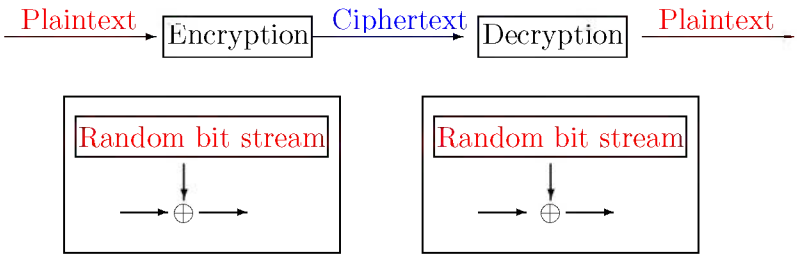
- full knowledge of the encryption/decryption algorithm,
- a number of plaintext/ciphertext pairs associated to the target key k.

If the number of possible keys is small then an attacker can break the system using an exhaustive search. The attacker encrypts one of the given plaintexts under all possible keys and determines which key produces the given ciphertext. Hence, the key space needs to be large enough to avoid such an attack. It is commonly assumed that a computation taking 2^{80} steps will be infeasible for a number of years to come, hence the key space size should be at least 80 bits to avoid exhaustive search.

The cipher designer must play two roles, that of someone trying to break as well as create a cipher. These days, although there is a lot of theory behind the design of many ciphers, we still rely on symmetric ciphers which are just believed to be strong, rather than ones for which we know a reason why they are strong. All this means is that the best attempts of the most experienced cryptanalysts cannot break them. This should be compared with public key ciphers, where there is now a theory which allows us to reason about how strong a given cipher is (given some explicit computational assumption).

Fig. 1 describes a simple model for enciphering bits, which although simple is quite suited to practical implementations. The idea of this model is to apply a reversible operation to the plaintext

FIGURE 1. Simple model for enciphering bits



to produce the ciphertext, namely combining the plaintext with a 'random stream'. The recipient can recreate the original plaintext by applying the inverse operation, in this case by combining the ciphertext with the same random stream.

This is particularly efficient since we can use the simplest operation available on a computer, namely exclusive-or $\oplus$. We saw in Chapter 5 that if the key is different for every message and the key is as long as the message, then such a system can be shown to be perfectly secure, namely we have the one-time pad. However, the one-time pad is not practical in many situations.

- We would like to use a short key to encrypt a long message.
- We would like to reuse keys.

Modern symmetric ciphers allow both of these properties, but this is at the expense of losing our perfect secrecy property. The reason for doing this is because using a one-time pad produces horrendous key distribution problems. We shall see that even using reusable short keys also produces bad (but not as bad) key distribution problems.

There are a number of ways to attack a bulk cipher, some of which we outline below. We divide our discussion into passive and active attacks; a passive attack is generally easier to mount than an active attack.

- **Passive Attacks:** Here the adversary is only allowed to listen to encrypted messages. Then he attempts to break the cryptosystem by either recovering the key or determining some secret that the communicating parties did not want leaked. One common form of passive attack is that of traffic analysis, a technique borrowed from the army in World War I, where a sudden increase in radio traffic at a certain point on the Western Front would signal an imminent offensive.

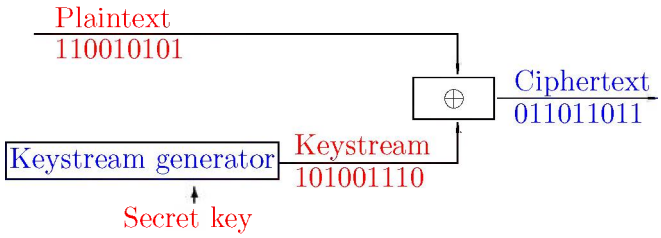
- **Active Attacks:** Here the adversary is allowed to insert, delete or replay messages between the two communicating parties. A general requirement is that an undetected insertion attack should require the breaking of the cipher, whilst the cipher needs to allow detection and recovery from deletion or replay attacks.

Bulk symmetric ciphers essentially come in two variants: stream ciphers, which operate on one data item (bit/letter) at a time, and block ciphers, which operate on data in blocks of items (e.g. 64 bits) at a time. In this chapter we look at stream ciphers, we leave block ciphers until Chapter 8.

2. Stream Cipher Basics

Fig. 2 gives a simple explanation of a stream cipher. Notice how this is very similar to our previous simple model. However, the random bit stream is now produced from a short secret key using a public algorithm, called the keystream generator.

FIGURE 2. Stream ciphers



Thus we have $c_i = m_i \oplus k_i$ where

- $m_0, m_1, \ldots$ are the plaintext bits,
- $k_0, k_1, \ldots$ are the keystream bits,
- $c_0, c_1, \ldots$ are the ciphertext bits.

This means

$$m_i = c_i \oplus k_i$$

i.e. decryption is the same operation as encryption.

Stream ciphers such as that described above are simple and fast to implement. They allow very fast encryption of large amounts of data, so they are suited to real-time audio and video signals. In addition there is no error propagation, if a single bit of ciphertext gets mangled during transit (due to an attacker or a poor radio signal) then only one bit of the decrypted plaintext will be affected. They are very similar to the Vernam cipher mentioned earlier, except now the key stream is only pseudo-random as opposed to truly random. Thus whilst similar to the Vernam cipher they are *not* perfectly secure.

Just like the Vernam cipher, stream ciphers suffer from the following problem; the same key used twice gives the same keystream, which can reveal relationships between messages. For example suppose m_1 and m_2 were encrypted under the same key k, then an adversary could work out the exclusive-or of the two plaintexts without knowing what the plaintexts were

$$c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k) = m_1 \oplus m_2.$$

Hence, there is a need to change keys frequently either on a per message or on a per session basis. This results in difficult key management and distribution techniques, which we shall see later how to solve using public key cryptography. Usually public key cryptography is used to determine

session or message keys, and then the actual data is rapidly encrypted using either a stream or block cipher.

The keystream generator above needs to produce a keystream with a number of properties for the stream cipher to be considered secure. As a bare minimum the keystream should

- Have a long period. Since the keystream k_i is produced via a deterministic process from the key, there will exist a number N such that
$$k_i = k_{i+N}$$
for all values of i. This number N is called the period of the sequence, and should be large for the keystream generator to be considered secure.
- Have pseudo-random properties. The generator should produce a sequence which appears to be random, in other words it should pass a number of statistical random number tests.
- Have large linear complexity. See Chapter 7 for what this means.

However, these conditions are not sufficient. Generally determining more of the sequence from a part should be computationally infeasible. Ideally, even if one knows the first one billion bits of the keystream sequence, the probability of guessing the next bit correctly should be no better than one half.

In Chapter 7 we shall discuss how stream ciphers are created using a combination of simple circuits called Linear Feedback Shift Registers. But first we will look at earlier constructions using rotor machines, or in modern notation Shift Registers (i.e. shift registers with no linear feedback).

3. The Lorenz Cipher

The Lorenz cipher was a German cipher from World War Two which was used for strategic information, as opposed to the tactical and battlefield information encrypted under the Enigma machine. The Lorenz machine was a stream cipher which worked on streams of bits. However it did not produce a single stream of bits, it produced five. The reason was due to the encoding of teleprinter messages used at the time, namely Baudot code.

3.1. Baudot Code. To understand the Lorenz cipher we first need to understand Baudot code. We all are aware of the ASCII encoding for the standard letters on a keyboard, this uses seven bits for the data, plus one bit for error detection. Prior to ASCII, indeed as far back as 1870, Baudot invented an encoding which used five bits of data. This was further developed until, by the 1930's, it was the standard method of communicating via teleprinter. The data was encoding via a tape, which consisted of a sequence of five rows of holes/non-holes.

Those of us of a certain age in the United Kingdom can remember the football scores being sent in on a Saturday evening by teleprinter, and those who are even older can maybe recall the ticker-tape parades in New York. The ticker-tape was the remains of transmitted messages in Baudot code. For those who can remember early dial-up modems, they will recall that the speeds were measured in Baud's, or characters per second, in memory of Baudot's invention.

Now five bits does not allow one to encode all the characters that one wants, thus Baudot code used two possible "states" called *letters shift* and *figures shift*. Moving between the two states was controlled by control characters, a number of other control characters were reserved for things such as space (SP), carriage return (CR), line feed (LF) or a character which rung the teleprinters bell (BELL) (such a code still exists in ASCII). The table for Baudot code in the 1930's is presented in Table 1.

Thus to transmit the message

Please, Please Help!

one would need to transmit the encoding, which we give in hexadecimal,

16, 12, 01, 03, 05, 01, 1B, 0C, 1F, 04, 16, 12, 01, 03, 05, 01, 04, 14, 01, 12, 16, 1B, 0D.

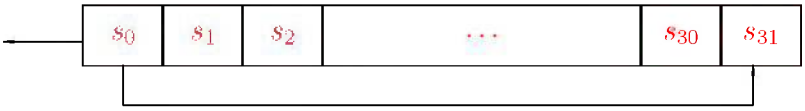
TABLE 1. The Baudot Code

Bits in Code					Letters Shift	Figures Shift
0	0	0	0	0	NULL	NULL
1	0	0	0	0	E	3
0	1	0	0	0	LF	LF
1	1	0	0	0	A	-
0	0	1	0	0	SP	SP
1	0	1	0	0	S	’
0	1	1	0	0	I	8
1	1	1	0	0	U	7
0	0	0	1	0	CR	CR
1	0	0	1	0	D	ENQ
0	1	0	1	0	R	4
1	1	0	1	0	J	BELL
0	0	1	1	0	N	,
1	0	1	1	0	F	!
0	1	1	1	0	C	:
1	1	1	1	0	K	(
0	0	0	0	1	T	5
1	0	0	0	1	Z	+
0	1	0	0	1	L	)
1	1	0	0	1	W	2
0	0	1	0	1	H	£
1	0	1	0	1	Y	6
0	1	1	0	1	P	0
1	1	1	0	1	Q	1
0	0	0	1	1	O	9
1	0	0	1	1	B	?
0	1	0	1	1	G	&
1	1	0	1	1	Figures	Figures
0	0	1	1	1	M	.
1	0	1	1	1	X	/
0	1	1	1	1	V	=
1	1	1	1	1	Letters	Letters

3.2. Lorenz Operation. The Lorenz cipher encrypted data in Baudot code form by producing a sequence of five random bits which was the exclusive or'd with the bits representing the Baudot code. The actual Lorenz cipher made use of a sequence of wheels, each wheel having a number of pins. The presence, or absence, of a pin signalling whether there was a one or zero signal. As the wheel turns, the position of the pins changes relative to an input signal. In modern parlance each wheel corresponds to a shift register,

Consider a register of length 32 bits, or equivalently a wheel with circumference 32. At each clock tick the register shifts left by one bit and the leftmost bit is output, alternatively the wheel turns around 1/32 of a revolution and the topmost pin is taken as the output of the wheel. This is represented in Figure 3. In Chapter 7 we shall see shift registers which have more complex feedback functions being used in modern stream ciphers, it is however interesting to see how similar ideas were used such a long time ago.

FIGURE 3. Shift Register of 32 bits



In Chapter 7 we shall see that the problem is how to combine the more complex shift registers into a secure cipher. The same problem exists with the Lorenz cipher, namely, how the relatively simple operation of the wheels/shift registers can be combined to produce a cipher which is hard to break. From now on we shall refer to these as shift registers as opposed to wheels.

A Lorenz cipher uses twelve registers to produce the five streams of random bits. The twelve registers are divided into three subsets. The first set consists of five shift registers which we denote by

$$\chi_j^{(i)}$$

by which we mean the output bit of the ith shift register on the jth clocking of the register. The five χ registers have lengths 41, 31, 29, 26 and 23, thus

$$\chi_{t+41}^{(1)} = \chi_t^{(1)}, \ \chi_{t+31}^{(2)} = \chi_t^{(2)}, \quad \text{etc}$$

for all values of t. The second set of five shift registers we denote by

$$\psi_j^{(i)}$$

for $i = 1, 2, 3, 4, 5$. These ψ registers have respective lengths 43, 47, 51, 53 and 59. The other two registers we shall denote by

$$\mu_j^{(i)}$$

for $i = 1, 2$, and are called the motor registers. The lengths of the μ registers are 61 and 37 respectively.

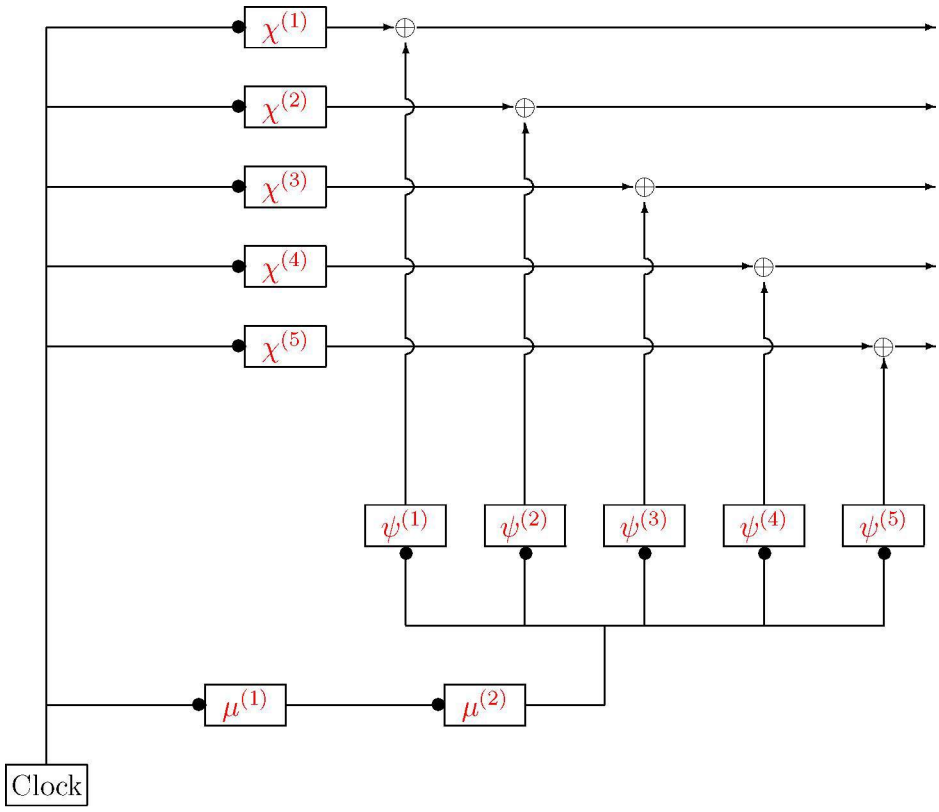
We now need to present how the Lorenz cipher clocks the various registers. To do this we use the variable t to denote a global clock, this will be ticked for every Baudot code character which is encrypted. We also use a variable t_ψ to denote how often the ψ registers have been clocked, and a variable t_μ which denotes how often the second μ register has been clocked. To start the cipher we set $t = t_\psi = t_\mu = 0$ but then these variables progress as follows: At a given point we perform the following operations:

(1) Let κ denote the vector $(\chi_t^{(i)} \oplus \psi_{t_\psi}^{(i)})_{i=1}^5$.
(2) If $\mu_{t+1}^{(1)} = 1$ then set $t_\mu = t_\mu + 1$.
(3) If $\mu_{t_\mu}^{(2)} = 1$ then set $t_\psi = t_\psi + 1$.
(4) Output κ.

The first line of the above produces the output keystream, the second line clocks the second μ register if the output of the first μ register is set (once it has been clocked), whilst the third line clocks all of the ψ registers if the output of the second μ register is set. From the above it should be deduced that the χ registers and the first μ register are clocked at every time interval. To encrypt a character the output vector κ is xor'd with the Baudot code representing the character of the plaintext. This is described graphically in Figure 4. In this figure we denote the clocking signal as a line with a circle on the end, output wires are denoted by an arrow.

The actual output of the ψ and μ motors at each time step are called the extended-ψ and the extended-μ streams. To ease future notation we will let $\psi'^{(i)}_t$ denote the output of the ψ register at

FIGURE 4. Graphical representation of the Lorenz cipher



time t, and the value $\mu'^{(2)}_t$ will denote the output of the second μ register at time t. In other words for a given tuple (t, t_ψ, t_μ) of valid clock values we have $\psi'^{(i)}_t = \psi_{t^{(i)}_\psi}$ and $\mu'^{(2)}_t = \mu^{(2)}_{t_\mu}$.

To see this in operation consider the following example: We denote the state of the cipher by the following notation

```
Chi:    111110001010110001111000101110100010001 11
        110000110101110110101101100 1000
        100010011110011000111011110 10
        111100011010001000111010 01
        110111100000010100011 10
Psi:    101100011010010100110101010101011010101 0100
        110101010101010111011010101010110001010101 01110
        101000010011010101010100010110101110101010100101001
        010101101010100001010100110110101001101101011010 11001
        010101010101010101101010010011010100100101010100100010010 10
Motor:  010111111011010110010001110111100010010011100011111010111 0100
        011101110001111111110000101011111111 1
```

This gives the states of the χ, ψ and μ registers at time $t = t_\psi = t_\mu = 0$. The states will be shifted leftwise, and the output of each register will be the left most bit. So executing the above

algorithm at time $t = 0$ the output first key vector will be

$$\kappa_0 = \chi_0 \oplus \psi_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}.$$

Then since $\mu_1^{(1)} = 1$ we clock the t_μ value, and then since $\mu_1^{(2)} = 1$ we also clock the t_ψ value. Thus at time $t = 1$ the state of the Lorenz cipher becomes

```
Chi:   11110001010110001111000101110100010001111
       10000110101110110101011001000 1
       00010011110011000111011110101
       11100011010001000111010011
       10111100000010100011101
Psi:   01100011010010100110101010101011010101001
       10101010101010111011010101010110001010101011101
       01000010011010101010100010110101110101010100101 0011
       10101101010100001010100110110101001101101011010110010
       10101010101010101101010010011010100100101010100100010010100
Motor: 10111111011010110010001110111100010010011100011111010111010 00
       11101110001111111110000101011111111110
```

Now we look at what happens at the next clock tick. At time $t = 1$ we now output the vector

$$\kappa_1 = \chi_1 \oplus \psi_0 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

But now since $\mu_{t+1}^{(1)}$ is equal to zero we do not clock t_μ, which means that since $\mu_1^{(2)} = 1$ we still clock the t_ψ value. This process is then repeated, so that we obtain the following sequence for the first 60 output values of the keystream κ_t,

```
0100100001010010111011000110110111011100011111110000000001001
0001000111011100111111010111110011000011011000111111101110111
0010100101100110111011101000010001001111001100101011010100 00
1010001011011100100110110010110001101000111100011111010101 11
1000110010000100010010000001010000001010001110000100110100 11
```

This is produced by xor'ing the output of the χ registers, which is given by

```
1111100010101100011110001011101000100011111111000101011000 11
1100001101011101101011011001000110000110101110110101101100 10
1000100111100110001110111101010001001111001100011101111010 10
1111000110100010001110100111110001101000100011101001111100 01
1101111000000101000111011011110000001010001110110111100000 01
```

by the values of output of the ψ'_t stream at time t,

```
1011000011111110100101001101011111111111000001101010110101 0
1101001010000001010101110110100000000000011111001010110001 01
1010000010000000110101010101000000000000000000011011010111 010
0101001101111110101000010101000000000000011111111011010100 110
0101001010000001010101011010000000000000000011001101010010
```

To ease understanding we also present the output $\mu'^{(2)}_t$ which is

$$11110111100000111111111111111000000000001000010111111111111$$

Recall, a one in this stream means that the ψ registers are clocked whilst a zero implies they are not clocked. One can see this effect in the ψ'_t output given earlier.

Just like the Enigma machine the Lorenz cipher has a long term key setup and a short term per message setup. The long term is the state of each register. Thus it appears there are a total of

$$2^{41+31+29+26+23+43+47+51+53+59+61+37} = 2^{501}$$

states, although the actual number is slightly less than this due to a small constraint which we be introduced in a moment. In the early stages of the war the μ registers were changes on a daily basis, the χ registers were changed on a monthly basis and the ψ registers were changed on a monthly or quarterly basis. Thus, if the months settings had been broken, then the "day" key, "only" consisted of at most

$$2^{61+37} = 2^{98}$$

states. As the war progressed the Germans moved to changing all the internal states of the registers every day.

Then, given these "day" values for the register contents, the per message setting is given by the starting position of each register. Thus the total number of message keys, given a day key, is given by

$$41 \cdot 31 \cdot 29 \cdot 26 \cdot 23 \cdot 43 \cdot 47 \cdot 51 \cdot 53 \cdot 59 \cdot 61 \cdot 37 \approx 2^{64}.$$

The Lorenz cipher has an obvious weakness as defined, which is what eventually led to its breaking, and which the Germans were aware of. The basic technique which we will use throughout the rest of this chapter is to take the 'Delta' of a sequence, this is defined as follows, for a sequence $s = (s_i)_{i=0}^{\infty}$,

$$\Delta s = (s_i \oplus s_{i+1})_{i=0}^{\infty}.$$

We shall denote the value of the Δs sequence at time t by $(\Delta s)_t$. The reason why the Δ operator is so important in the analysis of the Lorenz cipher is due to the following observation: Since

$$\kappa_t = \chi_t \oplus \psi_{t_\psi}$$

and

$$\kappa_{t+1} = \chi_{t+1} \oplus \left(\mu'^{(2)}_t \cdot \psi_{t_\psi+1}\right) \oplus \left((\mu'^{(2)}_t - 1) \cdot \psi_{t_\psi}\right),$$

thus

$$\begin{aligned}(\Delta\kappa)_t &= (\chi_t \oplus \chi_{t+1}) \oplus \left(\mu'^{(2)}_t \cdot \left(\psi_{t_\psi} \oplus \psi_{t_\psi+1}\right)\right) \\ &= (\Delta\chi)_t \oplus \left(\mu'^{(2)}_t \cdot (\Delta\psi)_{t_\psi}\right).\end{aligned}$$

Now if $\Pr[\mu'^{(2)}_t = 1] = \Pr[(\Delta_\psi)_{t_\psi} = 1] = 1/2$, as we would have by choosing the register states uniformly at random, then with probability $3/4$ the value of the $\Delta\kappa$ stream reveals the value of the $\Delta\chi$ stream, which would enable the adversary to recover the state of the χ registers relatively easily. Thus the Germans imposed a restriction on the key values so that

$$\Pr[\mu'^{(2)}_t = 1] \cdot \Pr[(\Delta_\psi)_{t_\psi} = 1] \approx 1/2.$$

In what follows we shall denote these two probabilities by $\delta = \Pr[\mu'^{(2)}_t = 1]$ and $\epsilon = \Pr[(\Delta_\psi)_{t_\psi} = 1]$.

Finally, to fix notation, if we let the Baudot encoding of the message be given by the sequence ϕ of 5-bit vectors, and the ciphertext be given by the sequence γ then we have

$$\gamma^{(i)}_t = \phi^{(i)}_t \oplus \kappa^{(i)}_t.$$

As the war progressed more complex internal operations of the Lorenz cipher were introduced. These were called "limitations" by Bletchley, and they introduced extra complications into the clocking of the various registers. We shall ignore these extra complications however in our discussion.

Initially the Allies did not know anything about the Lorenz cipher, even that it consisted of twelve wheels, let alone their period. In August 1941 the Germans made a serious mistake they transmitted virtually identical 4000 character messages using the exactly the same key. From this the cryptanalyst J. Tiltman managed to reconstruct the 4000 character key that had been output by the Lorenz cipher. From this sequence of 4000 apparently random strings of five bits another cryptographer W.T. Tutte recovered the precise internal workings of the Lorenz cipher. The final confirmation that the internal workings had been deduced correctly did not come until the end of the war, when the allies obtained a Lorenz machine on entering Germany.

3.3. Breaking the Wheels. Having determined the structure of the Lorenz cipher the problem remains on how to break it. The attack method were broken into two stages. In the first stage the wheel's needed to be broken, this was an involved process which only had to be performed once for each wheel configuration. Then a simpler procedure was produced which recovered the wheel positions for each message.

We now explain how wheel breaking occurred. The first task is to obtain with reasonable certainty the value of the sequence

$$\Delta\kappa^{(i)} \oplus \Delta\kappa^{(j)}$$

for different distinct values of i and j, usually $i = 1$ and $j = 2$. There were various different ways of performing this, below we present a gross simplification of the techniques used by the cryptanalysts at Bletchley. Our goal is simply to show that breaking even a 60 year old stream cipher requires some intricate manipulation of probability estimates, and that even small deviations from randomness in the output stream can cause a catastrophic failure in security.

To do this we first need to consider some characteristics of the plain text. Standard natural language contains a larger sequence of repeated characters than one would normally expect, compared to the case when a message was just random gibberish. If messages where random then one would expect

$$\Pr[(\Delta\phi^{(i)})_t \oplus (\Delta\phi^{(j)})_t = 0] = 1/2.$$

However, if the plaintext sequence contains slightly more repeated characters then we expect this probability to be slightly more than $1/2$, so we set

$$\Pr[(\Delta\phi^{(i)})_t \oplus (\Delta\phi^{(j)})_t = 0] = 1/2 + \rho. \tag{9}$$

Due to the nature of military German, and the Baudot encoding method, this was apparently particularly pronounced when one considered the first and second stream of bits, i.e. $i = 1$ and $j = 2$.

There are essentially two situations for wheel breaking, the first (more complex) case is when we do not know the underlying plaintext for a message, i.e. the attacker only has access to the ciphertext. The second case is when the attacker can guess with reasonable certainty the value of the underlying plaintext (a "crib" in the Bletchley jargon), and so can obtain the resulting keystream.

Ciphertext Only Method: The basic idea is that the sequence of ciphertext Delta's,

$$\Delta\gamma^{(i)} \oplus \Delta\gamma^{(j)}$$

will "reveal" the true value of the sequence

$$\Delta\chi^{(i)} \oplus \Delta\chi^{(j)}.$$

Consider the probability that we have

$$(\Delta\gamma^{(i)})_t \oplus (\Delta\gamma^{(j)})_t = (\Delta\chi^{(i)})_t \oplus (\Delta\chi^{(j)})_t. \tag{10}$$

Because of the relationship

$$\begin{aligned}(\Delta\gamma^{(i)})_t \oplus (\Delta\gamma^{(j)})_t &= (\Delta\phi^{(i)})_t \oplus (\Delta\phi^{(j)})_t \oplus (\Delta\kappa^{(i)})_t \oplus (\Delta\kappa^{(j)})_t \\ &= (\Delta\phi^{(i)})_t \oplus (\Delta\phi^{(j)})_t \oplus (\Delta\chi^{(i)})_t \oplus (\Delta\chi^{(j)})_t \\ &\quad \oplus \left(\mu'^{(2)}_t \cdot \left((\Delta\psi^{(i)})_{t_\psi} \oplus (\Delta\psi^{(j)})_{t_\psi}\right)\right),\end{aligned}$$

Equation 10 can hold in one of two ways;

- Either we have
$$(\Delta\phi^{(i)})_t \oplus (\Delta\phi^{(j)})_t = 0$$
and
$$\mu'^{(2)}_t \cdot \left((\Delta\psi^{(i)})_{t_\psi} \oplus (\Delta\psi^{(j)})_{t_\psi}\right) = 0.$$
The first of these events occurs with probability $1/2 + \rho$ by Equation 9, whilst the second occurs with probability
$$(1-\delta) + \delta \cdot (\epsilon^2 + (1-\epsilon)^2) = 1 - 2 \cdot \epsilon \cdot \delta + 2 \cdot \epsilon^2 \cdot \delta.$$
- Or we have
$$(\Delta\phi^{(i)})_t \oplus (\Delta\phi^{(j)})_t = 1$$
and
$$\mu'^{(2)}_t \cdot \left((\Delta\psi^{(i)})_{t_\psi} \oplus (\Delta\psi^{(j)})_{t_\psi}\right) = 1.$$
The first of these events occurs with probability $1/2 - \rho$ by Equation 9, whilst the second occurs with probability
$$2 \cdot \delta \cdot \epsilon \cdot (1-\epsilon).$$

Combining these probabilities together we find that Equation 10 holds with probability

$$\begin{aligned}&(1/2+\rho) \cdot (1 - 2 \cdot \epsilon \cdot \delta + 2 \cdot \epsilon^2 \cdot \delta) + 2 \cdot (1/2 - \rho) \cdot \delta \cdot \epsilon \cdot (1-\epsilon) \\ &\quad \approx (1/2+\rho)\epsilon + (1/2-\rho)(1-\epsilon) \\ &\quad = 1/2 + \rho \cdot (2 \cdot \epsilon - 1).\end{aligned}$$

since $\delta \cdot \epsilon \approx 1/2$ due to the key generation method mentioned earlier.

So assuming we have a sequence of n ciphertext characters, if we are trying to determine

$$\sigma_t = (\Delta\chi^{(1)})_t \oplus (\Delta\chi^{(2)})_t$$

i.e. we have set $i = 1$ and $j = 2$, then we know that this latter sequence has period $1271 = 41 \cdot 31$. Thus each element in this sequence will occur $n/1271$ times. If n is large enough, then taking a majority verdict will determine the value of the sequence σ_t with some certainty.

Known Keystream Method: Now assume that we know the value of $\kappa^{(i)}_t$. We use a similar idea to above, but now we use the sequence of keystream Delta's,

$$\Delta\kappa^{(i)} \oplus \Delta\kappa^{(j)}$$

and hope that this reveals the true value of the sequence

$$\Delta\chi^{(i)} \oplus \Delta\chi^{(j)}.$$

This is likely to happen due to the identity

$$(\Delta\kappa^{(i)})_t \oplus (\Delta\kappa^{(j)})_t = (\Delta\chi^{(i)})_t \oplus (\Delta\chi^{(j)})_t \oplus \left(\mu'^{(2)}_t \cdot \left((\Delta\psi^{(i)})_{t_\psi} \oplus (\Delta\psi^{(j)})_{t_\psi}\right)\right).$$

Hence we will have

$$(\Delta\kappa^{(i)})_t \oplus (\Delta\kappa^{(j)})_t = (\Delta\chi^{(i)})_t \oplus (\Delta\chi^{(j)})_t \tag{11}$$

precisely when

$$\mu'^{(2)}_t \cdot \left((\Delta\psi^{(i)})_{t_\psi} \oplus (\Delta\psi^{(j)})_{t_\psi} \right) = 0.$$

This last equation will hold with probability

$$\begin{aligned} (1-\delta) + \delta \cdot (\epsilon^2 + (1-\epsilon)^2) &= 1 - 2 \cdot \epsilon \cdot \delta + 2 \cdot \epsilon^2 \cdot \delta \\ &= 1 - 1 + \epsilon = \epsilon, \end{aligned}$$

since $\delta \cdot \epsilon \approx 1/2$. But since $\delta \cdot \epsilon \approx 1/2$ we usually have $0.6 \leq \epsilon \leq 0.8$, thus Equation 11 holds with a reasonable probability. So as before we try to take a majority verdict to obtain an estimate for each of the 1271 terms of the σ_t sequence.

Both Methods Continued: Which ever of the above methods we use there will still be some errors in our guess for the stream σ_t, which we will now try to correct. In some sense we are not really after the values for the sequence σ_t, what we really want is the exact values of the two shorter sequences

$$(\Delta\chi^{(1)})_t \text{ and } (\Delta\chi^{(2)})_t,$$

since these will allow us to deduce possible values for the first two χ registers in the Lorenz cipher. The approximation for the σ_t sequence of 1271 bits we now write down in a $41 \cdot 41$ bit array. By writing the first 31 bits into row one, the second 31 bits into row two and so on. A blank is placed into the array if we cannot determine the value of this bit with any reasonable certainty. For example, assuming the above configuration was used to encrypt the ciphertext, we could obtain an array which looks something like this;

```
0-0---01--1--110--110-10--1-0-1
010---0---10011-1----1-010-1--1
--00-1--111001--1-1--1101--1001
0-00--01----0-------0-10-0--001
-0-110-000-110-1000------1---10
-1--0---1-100110111-01---01---1
01-0-10111-001101-------1-1-0--
-01--0-0--0-100-00001-0---0----
1--1--10000-------0------1--11-
101---1-00-110--0-00--0-0---100
1---11--000---0---0-1--1------1
---1-0----011--1----1---01-01-0
-1---1--1--00110-1-1-110-0-100-
1-------10--10-1---0100-010--1-
0--0--011--0011--11-011------0-
--0001-1-11--11011---1-010110--
----10---------1000--001-10----
0-00-1-11110----1111-1-01-11-0-
----01-1-----11--111011-1--10--
0-0-01--1---011---11--1-1--1001
10-1---0-1-1-0010--01-0--10--10
----------1-01-0-1--0-10--1-001
010-01-1-110011-11---1-0---1--1
1-1-1-1000-11-010--01-0101-01-0
1---10-00--110---0-0--011-00-10
```

```
10-1-010-0----01----1--10-0----
-1--0-011-1001-0----01101011--1
010-010-11-0--101--10--0--1--0-
-01---1-0---1-01000-1---0-001-0
--1-10--0--110-100001-0--10-110
----1--00001-00--00010010----10
011-0101-110-1-011-101101----01
01---1----100--01---01-01--0-01
-011--1-000-1--10-00-0---1001-0
1011-01--0---001---01-01--0----
---0--0-1-1--11----1----1-11---
----0---1--0---0-----11--0--00-
10--101--0----0-0-0--0--010----
-100---1-110-----11-01-0---1---
0100----1-1----01-1--1111--11--
0-0001-1-1-0011011-1---01011-0-
```

Now the goal of the attacker is to fill in this array, a process known at Bletchley as "rectangling", noting that some of the zero's and one's entered could themselves be incorrect. The point to note is that when completed there should be just two distinct rows in the table, and each one should be the compliment of each other. A reasonable method to proceed is to take all rows starting with zero and then count the number of zeros and ones in the second element of those rows. We find there are seven ones and no zeros, doing the same for rows starting with a one we find there are four zeros and no ones. Thus we can deduce that the two types of rows in the table should start with a `10` and a `01`. Thus we then fill in the second element in any row which has its first element set. We continue in this way, first looking at rows and then looking at columns, until the whole table is filled in.

The above table was found using a few thousand characters of known keystream, which allows with the above method the simple reconstruction of the full table. According to the Bletchley documents the cryptographers at Bletchley would actually use a few hundred characters of keystream in a known keystream attack, and a few thousand in an unknown keystream attack. Since we are following rather naive methods our results are not as spectacular.

Once completed we can take the first column as the value of the $(\Delta\chi^{(1)})_t$ sequence and the first row as the value of the $(\Delta\chi^{(2)})_t$ sequence. We can then repeat this analysis for different pairs of the χ registers until we determine that we have

$$\begin{aligned}
\Delta\chi^{(1)} &= 000010011111010010001001110011100110011001000,\\
\Delta\chi^{(2)} &= 010001011110011011110110101100 1,\\
\Delta\chi^{(3)} &= 100110100010101001001100011111,\\
\Delta\chi^{(4)} &= 000100101110011001001110 10,\\
\Delta\chi^{(5)} &= 011000100000111100100 11.
\end{aligned}$$

From these $\Delta\chi$ sequences we can then determine possible values for the internal state of the χ registers.

So having "broken" the χ wheels of the Lorenz cipher, the task remains to determine the internal state of the other registers. In the ciphertext only attack one now needs to recover the actual keystream, a step which is clearly not needed in the known-keystream scenario. The trick here is to use the statistics of the underlying language again to try and recover the actual $\kappa^{(i)}$ sequence. We first de-χ the ciphertext sequence γ, using the values of the χ registers which we

have just determined, to obtain

$$\begin{aligned}\beta_t^{(i)} &= \gamma_t^{(i)} \oplus \chi_t^{(i)} \\ &= \phi_t^{(i)} \oplus \psi'^{(i)}_t.\end{aligned}$$

We then take the Δ of this β sequence

$$(\Delta\beta)_t = (\Delta\phi)_t \oplus \left(\mu'^{(2)}_t \cdot (\Delta\psi)_{t_\psi}\right),$$

and by our previous argument we will see that many values of the $\Delta\phi$ sequence will be "exposed" in the $\Delta\beta$ sequence. Using the knowledge of the $\Delta\phi$ sequence, e.g. it uses Baudot codes and natural language has many sequences of bigrams (e.g. space always following full stop), one can eventually recover the sequence ϕ and hence κ. At Bletchley this last step was usually performed by hand.

So in both scenarios we now have determined both the χ and the κ sequences. But what we are really after was the initial value of the registers ψ and μ. To determine these we de-χ the resulting κ sequence to obtain the ψ'_t sequence. In our example this would reveal the sequence

```
10110000111111101001010011010111111111111000001101010110101
11010010100000010101011101101000000000000011111001010110001 01
10100001000000011010101010100000000000000000011011010111010
01010011011111101010000101010000000000000111111011010100110
01010010100000010101010110101000000000000000001100110101001 0
```

given earlier. From this we can then recover a guess as to the $\mu'^{(2)}_t$ sequence.

```
11110111100000111111111111111000000000001000010111111111111...
```

Note, this is only a guess since it might occur that $\psi_{t_\psi} = \psi_{t_\psi+1}$, but we shall ignore this possibility. Once we have determined enough of the $\mu'^{(2)}_t$ sequence so that we have 59 ones in it, then we will have determined the initial state of the ψ registers. This is because after 59 clock ticks of the ψ registers all outputs have been presented in the ψ' sequence, since the largest ψ register has size 59.

All that remains is to determine the state of the μ registers. To do this we notice that the $\mu'^{(2)}_t$ sequence will make a transition from a 0 to a 1, or a 1 to a 0, precisely when $\mu^{(1)}_t$ outputs a one. By constructing enough of the $\mu'^{(2)}_t$ stream as above (say a few hundred bits) this allows us to determine the value of $\mu^{(1)}$ register almost exactly. Having recovered $\mu^{(1)}_t$ we can then deduce the values which must be contained in $\mu^{(2)}_{t_\mu}$ from this sequence and the resulting value of $\mu'^{(2)}_t$.

According to various documents in the early stages of the Lorenz cipher breaking effort at Bletchley, the entire "Wheel Breaking" operation was performed by hand. However, as time progressed the part which involved determining the $\Delta\chi$ sequences above from the rectangling procedure was eventually performed by the Colossus computer.

3.4. Breaking a Message. The Colossus computer was originally created not to break the wheels, i.e. to determine the longterm key of the Lorenz cipher. The Colossus was originally built to determine the per message settings, and hence to help break the individual ciphertexts. Whilst the previous method for breaking the wheels could be used to attack any ciphertext, to make it work efficiently you require a large ciphertext and a lot of luck. However, once the wheels are broken, i.e. we know the bits in the various registers, breaking the next ciphertext becomes easier.

Again we use the trick of de-χ'ing the ciphertext sequence γ, and then applying the Δ method to the resulting sequence β. We assume we know the internal states of all the registers but not their starting positions. We shall let s_i denote the unknown values of the starting positions of the five χ

wheels and s_ϕ (resp. s_μ) the global unknown starting position of the set of ϕ (resp. μ) wheels.

$$\begin{aligned}\beta_t &= \gamma_t \oplus \chi_{t+s_p} \\ &= \phi_t \oplus \psi'_{t+s_\phi};\end{aligned}$$

and then

$$(\Delta\beta)_t = (\Delta\phi)_{t+s_\phi} \oplus \left(\mu'^{(2)}_{t+s_\mu} \cdot (\Delta\psi)_{t_\psi}\right).$$

We then take two of the resulting five bit streams and xor them together as before to obtain

$$\begin{aligned}(\alpha^{(i,j)})_t &= (\Delta\beta^{(i)})_t \oplus (\Delta\beta^{(j)})_t \\ &= (\Delta\phi^{(i)})_{t+s_\phi} \oplus (\Delta\phi^{(j)})_{t+s_\phi} \oplus \mu'^{(2)}_{t+t_\mu}\left((\Delta\psi^{(i)})_{t_\psi} \oplus (\Delta\psi^{(j)})_{t_\psi}\right).\end{aligned}$$

Using our prior probability estimates we can determine the following probability estimate

$$\Pr[(\alpha^{(i,j)})_t = 0] \approx 1/2 + \rho \cdot (2 \cdot \epsilon - 1),$$

which is exactly the same probability we had for Equation 9 holding. In particular we note that $\Pr[\alpha^{(i,j)}{}_t = 0] > 1/2$, which forms the basis of this method of breaking into Lorenz ciphertexts.

Lets fix on $i = 1$ and $j = 2$. On assuming we know the values for the registers, all we need do is determine their starting positions s_1, s_2. We simply need to go through all $1271 = 41 \cdot 31$ possible starting positions for the first and second χ register. For each one of these starting positions we compute the associated $(\alpha^{(1,2)})_t$ sequence and count the number of values which are zero. Since we have $\Pr[\alpha_t^{(i,j)} = 0] > 1/2$ the correct value for the starting positions will correspond to a particularly high value for the count of the number of zeros.

This is a simple statistical test which allows one to determine the start positions of the first and second χ registers. Repeating this for other pairs of registers, or using similar statistical techniques, we can recover the start position of all χ registers. These statistical techniques are what the Colossus computer was designed to perform.

Once the χ register positions have been determined, the determination of the start positions of the ψ and μ registers was then performed by hand. The techniques for this are very similar to the earlier techniques needed to break the wheels, however once again various simplifications occur since one is assumed to know the state of each register, but not its start position.

Chapter Summary

- We have described the general model for symmetric ciphers, and stream ciphers in particular.
- We have looked at the Vernam cipher as a stream cipher, and described it's inner workings in terms of shift registers.
- We sketched how the Lorenz cipher was eventually broken. In particular you should notice that very tiny deviations from true randomness in the output can be exploited by a cryptographer in breaking a stream cipher.

Further Reading

The paper by Carter provides a more detailed description of the cryptanalysis performed at Bletchley on the Lorenz cipher. The book by Gannon is a very readable account of the entire

operation related to the Lorenz cipher, from obtaining the signals through to the construction and operation of the Colossus computer. For the "real" details you should consult the General Report on Tunny.

F.L. Carter. *The Breaking of the Lorenz Cipher: An Introduction to the Theory Behind the Operational Role of "Colossus" at BP.* In Coding and Cryptography - 1997, Springer-Verlag LNCS 1355, 74–88, 1997.

P. Gannon *Colossus: Bletchley Park's Greatest Secret.* Atlantic Books, 2007.

J. Good, D. Michie and G. Timms. *General report on Tunny, With Emphasis on Statistical Methods.* Document reference HW 25/4 and HW 25/5, Public Record Office Kew. Originally written in 1945, declassified in 2000.

CHAPTER 7

Modern Stream Ciphers

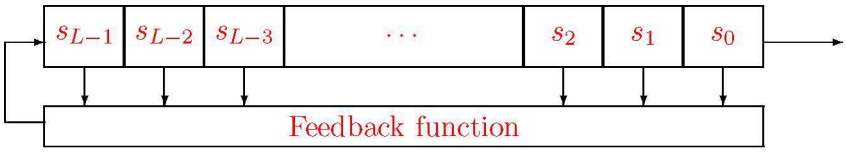
Chapter Goals

- To understand the basic principles of modern symmetric ciphers.
- To explain the basic workings of a modern stream cipher.
- To investigate the properties of linear feedback shift registers (LFSRs).

1. Linear Feedback Shift Registers

A standard way of producing a binary stream of data is to use a feedback shift register. These are small circuits containing a number of memory cells, each of which holds one bit of information. The set of such cells forms a register. In each cycle a certain predefined set of cells are 'tapped' and their value is passed through a function, called the *feedback function*. The register is then shifted down by one bit, with the output bit of the feedback shift register being the bit that is shifted out of the register. The combination of the tapped bits is then fed into the empty cell at the top of the register. This is explained in Fig. 1.

FIGURE 1. Feedback shift register



It is desirable, for reasons we shall see later, to use some form of non-linear function as the feedback function. However, this is often hard to do in practice hence usually one uses a linear feedback shift register, or LFSR for short, where the feedback function is a linear function of the tapped bits. In each cycle a certain predefined set of cells are 'tapped' and their value is XORed together. The register is then shifted down by one bit, with the output bit of the LFSR being the bit that is shifted out of the register. Again, the combination of the tapped bits is then fed into the empty cell at the top of the register.

Mathematically this can be defined as follows, where the register is assumed to be of length L. One defines a set of bits $[c_1, \ldots, c_L]$ which are set to one if that cell is tapped and set to zero otherwise. The initial internal state of the register is given by the bit sequence $[s_{L-1}, \ldots, s_1, s_0]$. The output sequence is then defined to be $s_0, s_1, s_2, \ldots, s_{L-1}, s_L, s_{L+1}, \ldots$ where for $j \geq L$ we have

$$s_j = c_1 \cdot s_{j-1} \oplus c_2 \cdot s_{j-2} \oplus \cdots \oplus c_L \cdot s_{j-L}.$$

Note, that for an initial state of all zeros the output sequence will be the zero sequence, but for a non-zero initial state the output sequence must be eventually periodic (since we must eventually

return to a state we have already been in). The period of a sequence is defined to be the smallest integer N such that

$$s_{N+i} = s_i$$

for all sufficiently large i. In fact there are $2^L - 1$ possible non-zero states and so the most one can hope for is that an LFSR, for all non-zero initial states, produces an output stream whose period is of length exactly $2^L - 1$.

Each state of the linear feedback shift register can be obtained from the previous state via a matrix multiplication. If we write

$$M = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ c_L & c_{L-1} & c_{L-2} & \dots & c_1 \end{pmatrix}$$

and

$$v = (1, 0, 0, \dots, 0)$$

and we write the internal state as

$$s = (s_1, s_2, \dots, s_L)$$

then the next state can be deduced by computing

$$s = M \cdot s$$

and the output bit can be produced by computing the vector product

$$v \cdot s.$$

The properties of the output sequence are closely tied up with the properties of the binary polynomial

$$C(X) = 1 + c_1 X + c_2 X^2 + \cdots + c_L X^L \in \mathbb{F}_2[X],$$

called the connection polynomial for the LFSR. The connection polynomial and the matrix are related via

$$C(X) = \det(XM - I_L).$$

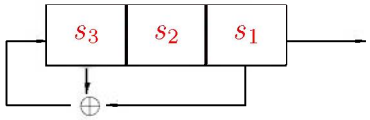
In some text books the connection polynomial is written in reverse, i.e. they use

$$G(X) = X^L C(1/X)$$

as the connection polynomial. One should note that in this case $G(X)$ is the characteristic polynomial of the matrix M.

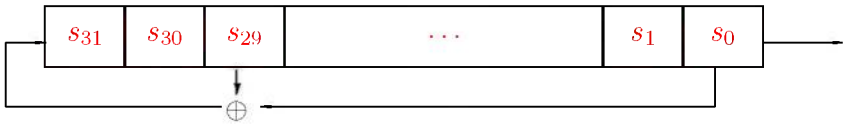
As an example see Fig. 2 for an LFSR in which the connection polynomial is given by X^3+X+1 and Fig. 3 for an LFSR in which the connection polynomial is given by $X^{32} + X^3 + 1$.

FIGURE 2. Linear feedback shift register: $X^3 + X + 1$



Of particular importance is when the connection polynomial is primitive.

FIGURE 3. Linear feedback shift register: $X^{32} + X^3 + 1$



DEFINITION 7.1. *A binary polynomial $C(X)$ of degree L is primitive if it is irreducible and a root θ of $C(X)$ generates the multiplicative group of the field $\mathbb{F}_{2^L}$. In other words, since $C(X)$ is irreducible we already have*

$$\mathbb{F}_2[X]/(C(X)) = \mathbb{F}_2(\theta) = \mathbb{F}_{2^L},$$

but we also require

$$\mathbb{F}_{2^L}^* = \langle\theta\rangle.$$

The properties of the output sequence of the LFSR can then be deduced from the following cases.

- $c_L = 0$:
  In this case the sequence is said to be singular. The output sequence may not be periodic, but it will be eventually periodic.
- $c_L = 1$:
  Such a sequence is called non-singular. The output is always purely periodic, in that it satisfies $s_{N+i} = s_i$ for all i rather than for all sufficiently large values of i. Of the non-singular sequences of particular interest are those satisfying
  - $C(X)$ is irreducible:
    Every non-zero initial state will produce a sequence with period equal to the smallest value of N such that $C(X)$ divides $1 + X^N$. We have that N will divide $2^L - 1$.
  - $C(X)$ is primitive:
    Every non-zero initial state produces an output sequence which is periodic and of exact period $2^L - 1$.

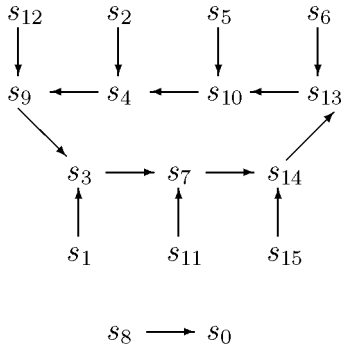
We do not prove these results here, but proofs can be found in any good textbook on the application of finite fields to coding theory, cryptography or communications science. However, we present four examples which show the different behaviours. All examples are on four bit registers, i.e. $L = 4$.

Example 1 : In this example we use an LFSR with connection polynomial $C(X) = X^3 + X + 1$. We therefore see that $\deg(C) \neq L$, and so the sequence will be singular. The matrix M generating the sequence is given by

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

If we label the states of the LFSR by the number whose binary representation is the state value, i.e. $s_0 = (0, 0, 0, 0)$ and $s_5 = (0, 1, 0, 1)$, then the periods of this LFSR can be represented by the transitions in Figure 4. Note, it is not purely periodic.

FIGURE 4. Transitions of the four bit LFSR with connection polynomial $X^3 + X + 1$

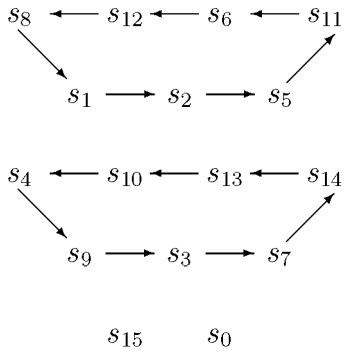


Example 2 : Now let the connection polynomial $C(X) = X^4+X^3+X^2+1 = (X+1)(X^3+X+1)$, which corresponds to the matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

The state transitions are then given by Figure 5. Note, it is purely periodic, but that there are different period lengths due to the different factorization properties of the connection polynomial modulo 2. One of length $7 = 2^3 - 1$ corresponding to the factor of degree three, and one of length $1 = 2^1 - 1$ corresponding to the factor of degree one. We ignore the trivial period of the zero'th state.

FIGURE 5. Transitions of the four bit LFSR with connection polynomial $X^4+X^3+X^2+1$

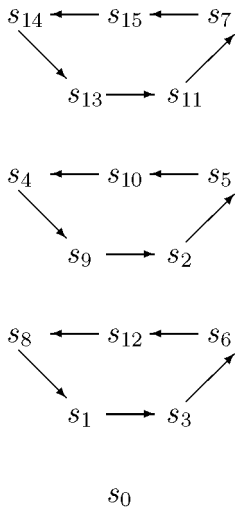


Example 3 : Now take the connection polynomial $C(X) = X^4 + X^3 + X^2 + X + 1$, which is irreducible, but not primitive. The matrix is now given by

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

The state transitions are then given by Figure 6. Note, it is purely periodic and all periods have same length, bar the trivial one.

FIGURE 6. Transitions of the four bit LFSR with connection polynomial $X^4+X^3+X^2+X+1$

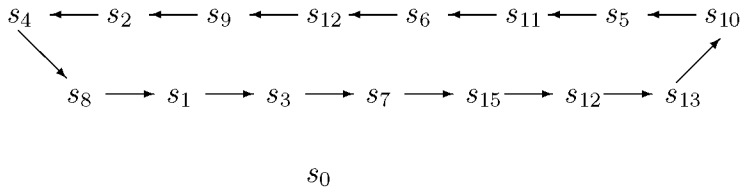


Example 4 : As our final example we take the connection polynomial $C(X) = X^4+X+1$, which is irreducible and primitive. The matrix M is now

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

and the state transitions are given by Figure 7.

FIGURE 7. Transitions of the four bit LFSR with connection polynomial X^4+X+1



Whilst there are algorithms to generate primitive polynomials for use in applications we shall not describe them here. We give some samples in the following list, where we give polynomials with a small number of taps for efficiency.

$$\begin{array}{ccc} x^{31}+x^3+1 & x^{31}+x^6+1 & x^{31}+x^7+1 \\ x^{39}+x^4+1 & x^{60}+x+1 & x^{63}+x+1 \\ x^{71}+x^6+1 & x^{93}+x^2+1 & x^{137}+x^{21}+1 \\ x^{145}+x^{52}+1 & x^{161}+x^{18}+1 & x^{521}+x^{32}+1 \end{array}$$

Although LFSRs efficiently produce bitstreams from a small key, especially when implemented in hardware, they are not usable on their own for cryptographic purposes. This is because they are essentially linear, which is after all why they are efficient.

We shall now show that if we know an LFSR has L internal registers, and we can determine $2L$ consecutive bits of the stream then we can determine the whole stream. First notice we need to determine L unknowns, the L values of the 'taps' c_i, since the L values of the initial state $s_0, \ldots, s_{L-1}$ are given to us. This type of data could be available in a known plaintext attack, where we obtain the ciphertext corresponding to a known piece of plaintext, since the encryption operation is simply exclusive-or we can determine as many bits of the keystream as we require.

Using the equation

$$s_j = \sum_{i=1}^{L} c_i \cdot s_{j-i} \pmod 2,$$

we obtain $2L$ linear equations, which we then solve via matrix techniques. We write our matrix equation as

$$\begin{pmatrix} s_{L-1} & s_{L-2} & \cdots & s_1 & s_0 \\ s_L & s_{L-1} & \cdots & s_2 & s_1 \\ \vdots & \vdots & & \vdots & \vdots \\ s_{2L-3} & s_{2L-4} & \cdots & s_{L-1} & s_{L-2} \\ s_{2L-2} & s_{2L-3} & \cdots & s_L & s_{L-1} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_{L-1} \\ c_L \end{pmatrix} = \begin{pmatrix} s_L \\ s_{L+1} \\ \vdots \\ s_{2L-2} \\ s_{2L-1} \end{pmatrix}.$$

As an example, suppose we see the output sequence

$$1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, \ldots$$

and we are told that this sequence was the output of a four-bit LFSR. Using the above matrix equation, and solving it modulo 2, we would find that the connection polynomial was given by

$$X^4 + X + 1.$$

Hence, we can conclude that a stream cipher based solely on a single LFSR is insecure against a known plaintext attack.

An important measure of the cryptographic quality of a sequence is given by the linear complexity of the sequence.

DEFINITION 7.2 (Linear complexity). *For an infinite binary sequence*

$$s = s_0, s_1, s_2, s_3, \ldots,$$

we define the linear complexity of s as $L(s)$ where

- *$L(s) = 0$ if s is the zero sequence,*
- *$L(s) = \infty$ if no LFSR generates s,*
- *$L(s)$ will be the length of the shortest LFSR to generate s.*

Since we cannot compute the linear complexity of an infinite set of bits we often restrict ourselves to a finite set s^n of the first n bits. The linear complexity satisfies the following properties for any sequence s.

- For all $n \geq 1$ we have $0 \leq L(s^n) \leq n$.
- If s is periodic with period N then $L(s) \leq N$.
- $L(s \oplus t) \leq L(s) + L(t)$.

For a random sequence of bits, which is what we want from a stream cipher's keystream generator, we should have that the expected linear complexity of s^n is approximately just larger than $n/2$. But for a keystream generated by an LFSR we know that we will have $L(s^n) = L$ for all $n \geq L$. Hence, an LFSR produces nothing at all like a random bit string.

We have seen that if we know the length of the LFSR then, from the output bits, we can generate the connection polynomial. To determine the length we use the linear complexity profile, this is defined to be the sequence $L(s^1), L(s^2), L(s^3), \ldots$. There is also an efficient algorithm called the Berlekamp–Massey algorithm which given a finite sequence s^n will compute the linear complexity profile

$$L(s^1), L(s^2), L(s^3), \ldots, L(s^n).$$

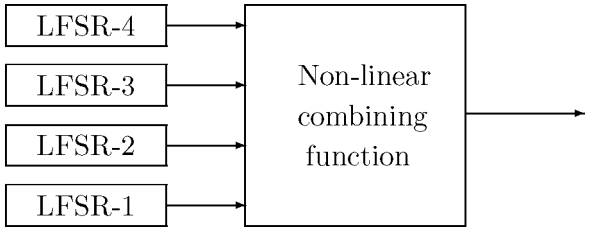
In addition the Berlekamp–Massey algorithm will also output the associated connection polynomial, if $n \geq L(s^n)/2$, using a technique more efficient than the prior matrix technique.

Hence, if we use an LFSR of size L to generate a keystream for a stream cipher and the adversary obtains at least $2L$ bits of this keystream then they can determine the exact LFSR used and so generate as much of the keystream as they wish. Therefore, one needs to find a way of using LFSRs in some non-linear way, which hides the linearity of the LFSRs and produces output sequences with high linear complexity.

2. Combining LFSRs

To use LFSRs in practice it is common for a number of them to be used, producing a set of output sequences $x_1^{(i)}, \ldots, x_n^{(i)}$. The key is then the initial state of all of the LFSRs and the keystream is produced from these n generators using a non-linear combination function $f(x_1, \ldots, x_n)$, as described in Fig. 8.

FIGURE 8. Combining LFSRs



We begin by examining the case where the combination function is a boolean function of the output bits of the constituent LFSRs. For analysis of this function we write it as a sum of distinct products of variables, e.g.

$$f(x_1, x_2, x_3, x_4, x_5) = 1 \oplus x_2 \oplus x_3 \oplus x_4 \cdot x_5 \oplus x_1 \cdot x_2 \cdot x_3 \cdot x_5.$$

However, in practice the boolean function could be implemented in a different way. When expressed as a sum of products of variables we say that the boolean function is in algebraic normal form.

Suppose that one uses n LFSRs of maximal length (i.e. all with a primitive connection polynomial) and whose periods $L_1, \ldots, L_n$ are all distinct and greater than two. Then the linear complexity of the keystream generated by $f(x_1, \ldots, x_n)$ is equal to

$$f(L_1, \ldots, L_n)$$

where we replace $\oplus$ in f with integer addition and multiplication modulo two by integer multiplication, assuming f is expressed in algebraic normal form. The non-linear order of the polynomial f is then equal to total the degree of f.

However, it turns out that creating a nonlinear function which results in a high linear complexity is not the whole story. For example consider the stream cipher produced by the Geffe generator.

This generator takes three LFSRs of maximal period and distinct sizes, L_1, L_2 and L_3, and then combines them using the non-linear function, of non-linear order 2,

$$z = f(x_1, x_2, x_3) = x_1 \cdot x_2 \oplus x_2 \cdot x_3 \oplus x_3. \tag{12}$$

This would appear to have very nice properties: It's linear complexity is given by

$$L_1 \cdot L_2 + L_2 \cdot L_3 + L_3$$

and it's period is given by

$$(2^{L_1} - 1)(2^{L_2} - 1)(2^{L_3} - 1).$$

However, it turns out to be cryptographically weak.

To understand the weakness of the Geffe generator consider the following table, which presents the outputs x_i of the constituent LFSRs and the resulting output z of the Geffe generator

x_1	x_2	x_3	z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

If the Geffe generator was using a "good" non-linear combining function then the output bits z would not reveal any information about the corresponding output bits of the constituent LFSRs. However, we can easily see that

$$\Pr(z = x_1) = 3/4 \text{ and } \Pr(z = x_3) = 3/4.$$

This means that the output bits of the Geffe generator are correlated with the bits of two of the constituent LFSRs. This means that we can attack the generator using a correlation attack.

This attack proceeds as follows, suppose we know the lengths L_i of the constituent generators, but not the connection polynomials or their initial states. The attack is desribed in Algorithm 7.1

Algorithm 7.1: Correlation attack on the Geffe generator

forall *primitive connection polynomials of degree* L_1 **do**
  forall *Initial states of the first LFSR* **do**
    Compute $2L_1$ bits of output of the first LFSR
    Compute how many are equal to the output of the Geffe generator
    A large value signals that this is the correct choice of generator and starting state.
  end
end
Repeat the above for the third LFSR
Recover the second LFSR by testing possible values using (12)

It turns out that their are a total of

$$S = \phi(2^{L_1} - 1) \cdot \phi(2^{L_2} - 1) \cdot \phi(2^{L_3} - 1)/(L_1 \cdot L_2 \cdot L_3)$$

possible connection polynomials for the three LFSRs in the Geffe generator. The total number of initial states of the Geffe generator is

$$T = (2^{L_1} - 1)(2^{L_2} - 1)(2^{L_3} - 1) \approx 2^{L_1 + L_2 + L_3}.$$

This means that the key size of the Geffe generator is

$$S \cdot T \approx S \cdot (2^{L_1+L_2+L_3}).$$

For a secure stream cipher we would like the size of the key space to be about the same as the number of operations needed to break the stream cipher. However, the above correlation attack on the Geffe generator requires roughly

$$S \cdot (2^{L_1} + 2^{L_2} + 2^{L_3})$$

operations. The reason for the reduced complexity is that we can deal with each constituent LFSR in turn.

To combine high linear complexity and resistance to correlation attacks (and other attacks) designers have had to be a little more ingenious as to how they have produced non-linear combiners for LFSRs. We now outline a small subset of some of the most influential:

Filter Generator: The basic idea here is to take a single primitive LFSR with internal state $s_1, \ldots, s_L$ and then make the output of the stream cipher be a non-linear function of the whole state, i.e.

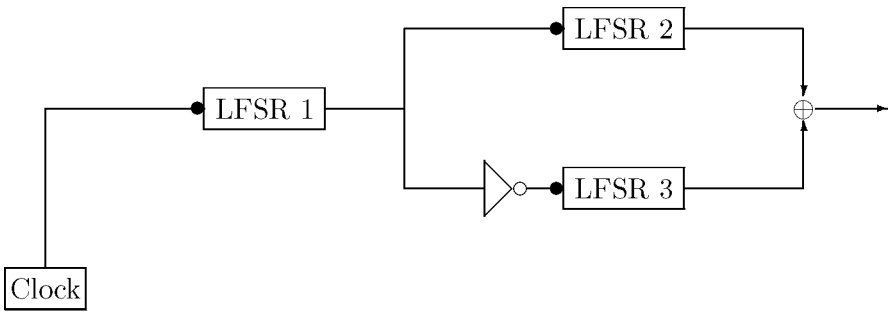
$$z = F(s_1, \ldots, s_L).$$

If F has non-linear order m then the linear complexity of the resulting sequence is given by

$$\sum_{i=1}^{m} \binom{L}{i}.$$

Alternating Step Generator: This takes three LFSRs of size L_1, L_2 and L_3 which are pairwise coprime, and of roughly the same size. If the output sequence of the three LFSRs is denoted by x_1, x_2 and x_3, then one proceeds as follows: The first LFSR is clocked on every iteration. If its output x_1 is equal to one, then the second LFSR is clocked and the output of the third LFSR is repeated from its last value. If the output of x_1 is equal to zero, then the third LFSR is clocked and the output of the second LFSR is repeated from its last value. The output of the generator is the value of $x_2 \oplus x_3$. This operation is described graphically in Figure 9.

FIGURE 9. Graphical representation of the Alternating step generator



The alternating step generator has period

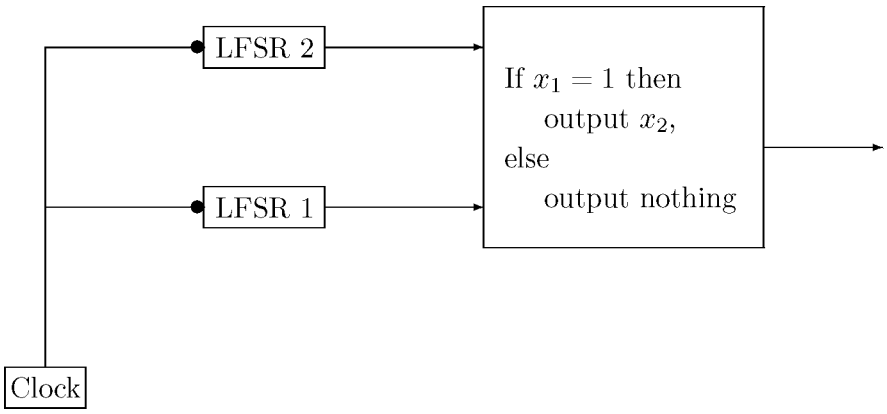
$$2^{L_1}(2^{L_2} - 1)(2^{L_3} - 1)$$

and linear complexity, approximately

$$(L_2 + L_3) \cdot 2^{L_1}.$$

Shrinking Generator: Here we take two LFSRs with output sequence x_1 and x_2, and the idea is to throw away some of the x_2 stream under the control of the x_1 stream. Both LFSRs are clocked at the same time, and if x_1 is equal to one then the output of the generator is the value of x_2. If x_1 is equal to zero then the generator just clocks again. Note, that this means that the generator does not produce a bit on each iteration. This operation is described graphically in Figure 10.

FIGURE 10. Graphical representation of the Shrinking Generator



If we assume that the two constituent LFSRs have size L_1 and L_2 with $\gcd(L_1, L_2)$ equal to one, then the period of the shrinking generator is equal to

$$(2^{L_2} - 1) \cdot 2^{L_1 - 1}$$

and its linear complexity is approximately

$$L_2 \cdot 2^{L_1}.$$

The A5/1 Generator: Probably the most famous of the recent LFSR based stream ciphers is A5/1. This is the stream cipher used to encrypt the on-air traffic in the GSM mobile phone networks in Europe and the US. This was developed in 1987, buts its design was kept secret until 1999 when it was reverse engineered. There is a weakened version of the algorithm called A5/2 which was designed for use in places where there were various export restrictions. In recent years various attacks have been published on A5/1 which has resulted in it no longer being considered a secure cipher. In the replacement for GSM, i.e. UMTS or 3G networks, the cipher has been replaced with the use of the block cipher KASUMI in a stream cipher mode of operation.

A5/1 makes use of three LFSRs of length 19, 22 and 23. These have characteristic polynomials

$$x^{18} + x^{17} + x^{16} + x^{13} + 1,$$
$$x^{21} + x^{20} + 1,$$
$$x^{22} + x^{21} + x^{20} + x^7 + 1.$$

Alternatively (and equivalently) their connection polynomials are given by

$$x^{18} + x^5 + x^2 + x^1 + 1,$$
$$x^{21} + x^1 + 1,$$
$$x^{22} + x^{15} + x^2 + x^1 + 1.$$

The output of the cipher is the exclusive-or of the three output bits of the three LFSRs.

To clock the registers we associate to each register a "clocking bit". These are in positions 10, 11 and 12 of the LFSR's (assuming bits are ordered with 0 corresponding to the output bit, other books may use a different ordering). We will call these bits c_1, c_2 and c_3. At each clock step the three bits are computed and the "majority bit" is determined via the formulae

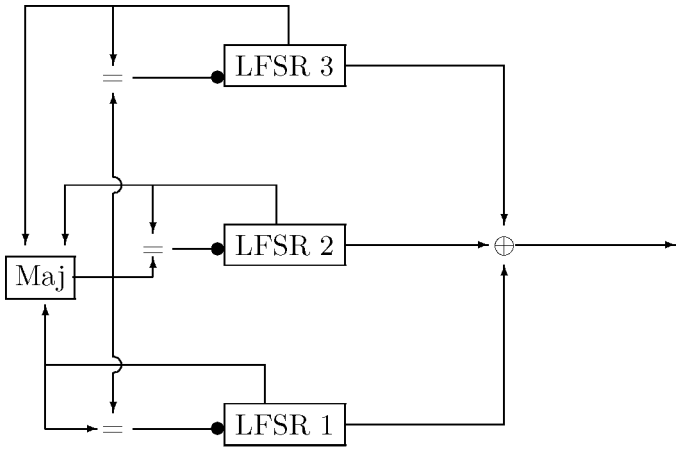
$$c_1 \cdot c_2 \oplus c_2 \cdot c_3 \oplus c_1 \cdot c_3.$$

The ith LFSR is then clocked if the majority bit is equal to the bit c_i. Thus clocking occurs subject to the following table

			Majority	Clock LFSR		
c_1	c_2	c_3	Bit	1	2	3
0	0	0	0	Y	Y	Y
0	0	1	0	Y	Y	N
0	1	0	0	Y	N	Y
0	1	1	1	N	Y	Y
1	0	0	0	N	Y	Y
1	0	1	1	Y	N	Y
1	1	0	1	Y	Y	N
1	1	1	1	Y	Y	Y

Thus we see in A5/1 that each LFSR is clocked with probability 3/4. This operation is described graphically in Figure 11.

FIGURE 11. Graphical representation of the A5/1 Generator



3. RC4

RC stands for Ron's Cipher after Ron Rivest of MIT. You should not think that the RC4 cipher is a prior version of the block ciphers RC5 and RC6. It is in fact a very, very fast stream cipher. It is very easy to remember since it is surprisingly simple.

Given an array S indexed from 0 to 255 consisting of the integers $0, \ldots, 255$, permuted in some key-dependent way, the output of the RC4 algorithm is a keystream of bytes K which is XORed with the plaintext byte by byte. Since the algorithm works on bytes and not bits, and uses very simple operations it is particularly fast in software. We start by letting $i = 0$ and $j = 0$, we then repeat the steps in Algorithm 7.2.

The security rests on the observation that even if the attacker knows K and i, he can deduce the value of S_t, but this does not allow him to deduce anything about the internal state of the

Algorithm 7.2: RC4 Algorithm

$i = (i + 1) \mod 256$
$j = (j + S_i) \mod 256$
$\text{swap}(S_i, S_j)$
$t = (S_i + S_j) \mod 256$
$K = S_t$

table. This follows from the observation that he cannot deduce the value of t, as he does not know j, S_i or S_j.

It is a very tightly designed algorithm as each line of the code needs to be there to make the cipher secure.

- $i = (i + 1) \mod 256$ **:**
  Makes sure every array element is used once after 256 iterations.
- $j = (j + S_i) \mod 256$ **:**
  Makes the output depend non-linearly on the array.
- **swap**(S_i, S_j) **:**
  Makes sure the array is evolved and modified as the iteration continues.
- $t = (S_i + S_j) \mod 256$ **:**
  Makes sure the output sequence reveals little about the internal state of the array.

The initial state of the array S is determined from the key using the method described by Algorithm 7.3.

Algorithm 7.3: RC4 Key Schedule

for $i = 0$ **to** 255 **do** $S_i = i$
Initialise K_i, for $i = 0, \ldots, 255$, with the key, repeating if neccesary
$j = 0$
for $i = 0$ **to** 255 **do**
  $j = (j + S_i + K_i) \mod 256$
  $\text{swap}(S_i, S_j)$
end

Although RC4 is very fast for a software based stream cipher, there are some issues with its use. In particular both the key schedule and the main algorithm do not produce as random a stream as one might wish. Hence, it should be used with care.

Chapter Summary

- Modern stream ciphers can be obtained by combining, in a non-linear way, simple bit generators called LFSRs, these stream ciphers are bit oriented.
- LFSR based stream ciphers provide very fast ciphers, suitable for implementation in hardware, which can encrypt real-time data such as voice or video.
- RC4 provides a fast and compact byte oriented stream cipher for use in software.

Further Reading

A good introduction to linear recurrence sequences over finite fields is in the book by Lidl and Neiderreiter. This book covers all the theory one requires, including examples and a description of the Berlekamp–Massey algorithm. Analysis of the RC4 algorithm can be found in the Master's thesis of Mantin.

R. Lidl and H. Neiderreiter. *Introduction to fintie field and their applications.* Cambridge University Press, 1986.

I. Mantin. *Analysis of the Stream Cipher RC4.* MSc. Thesis, Weizmann Institute of Science, 2001.

CHAPTER 8

Block Ciphers

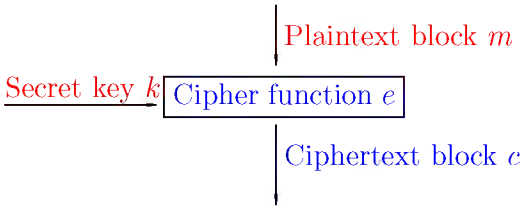
Chapter Goals

- To introduce the notion of block ciphers.
- To understand the workings of the DES algorithm.
- To understand the workings of the Rijndael algorithm.
- To learn about the various standard modes of operation of block ciphers.

1. Introduction To Block Ciphers

The basic description of a block cipher is shown in Fig. 1. Block ciphers operate on blocks

FIGURE 1. Operation of a block cipher



of plaintext one at a time to produce blocks of ciphertext. The main difference between a block cipher and a stream cipher is that block ciphers are stateless, whilst stream ciphers maintain an internal state which is needed to determine which part of the keystream should be generated next. We write

$$c = e_k(m),$$
$$m = d_k(c)$$

where

- m is the plaintext block,
- k is the secret key,
- e is the encryption function,
- d is the decryption function,
- c is the ciphertext block.

The block sizes taken are usually reasonably large, 64 bits in DES and 128 bits or more in modern block ciphers. Often the output of the ciphertext produced by encrypting the first block is used to help encrypt the second block in what is called a *mode of operation*. These modes are used to avoid certain attacks based on deletion or insertion by giving each ciphertext block a context within the overall message. Each mode of operation offers different protection against error propagation due to transmission errors in the ciphertext. In addition, depending on the mode of operation (and the application) message/session keys may be needed. For example, many modes require a per message

initial value to be input into the encryption and decryption operations. Later in this chapter we shall discuss modes of operation of block ciphers in more detail.

There are many block ciphers in use today, some which you may find used in your web browser are RC5, RC6, DES or 3DES. The most famous of these is DES, or the Data Encryption Standard. This was first published in the mid-1970s as a US Federal standard and soon become the de-facto international standard for banking applications.

The DES algorithm has stood up remarkably well to the test of time, but in the early 1990s it became clear that a new standard was required. This was because both the block length (64 bits) and the key length (56 bits) of basic DES were too small for future applications. It is now possible to recover a 56-bit DES key using either a network of computers or specialized hardware. In response to this problem the US National Institute for Standards and Technology (NIST) initiated a competition to find a new block cipher, to be called the Advanced Encryption Standard or AES.

Unlike the process used to design DES, which was kept essentially secret, the design of the AES was performed in public. A number of groups from around the world submitted designs for the AES. Eventually five algorithms, known as the AES finalists, were chosen to be studied in depth. These were

- MARS from a group at IBM,
- RC6 from a group at RSA Security,
- Twofish from a group based at Counterpane, UC Berkeley and elsewhere,
- Serpent from a group of three academics based in Israel, Norway and the UK,
- Rijndael from a couple of Belgian cryptographers.

Finally in the fall of 2000, NIST announced that the overall AES winner had been chosen to be Rijndael.

DES and all the AES finalists are examples of *iterated* block ciphers. The block ciphers obtain their security by repeated use of a simple *round function*. The round function takes an n-bit block and returns an n-bit block, where n is the block size of the overall cipher. The number of rounds r can either be a variable or fixed. As a general rule increasing the number of rounds will increase the level of security of the block cipher.

Each use of the round function employs a round key

$$k_i \text{ for } 1 \leq i \leq r$$

derived from the main secret key k, using an algorithm called a *key schedule*. To allow decryption, for every round key the function implementing the round must be invertible, and for decryption the round keys are used in the opposite order that they were used for encryption. That the whole round is invertible does not imply that the functions used to implement the round need to be invertible. This may seem strange at first reading but will become clearer when we discuss the DES cipher later. In DES the functions needed to implement the round function are not invertible, but the whole round is invertible. For Rijndael not only is the whole round function invertible but every function used to create the round function is also invertible.

There are a number of general purpose techniques which can be used to break a block cipher, for example: exhaustive search, using pre-computed tables of intermediate values or divide and conquer. Some (badly designed) block ciphers can be susceptible to chosen plaintext attacks, where encrypting a specially chosen plaintext can reveal properties of the underlying secret key. In cryptanalysis one needs a combination of mathematical and puzzle-solving skills, plus luck. There are a few more advanced techniques which can be employed, some of which apply in general to any cipher (and not just a block cipher).

- **Differential Cryptanalysis:** In differential cryptanalysis one looks at ciphertext pairs, where the plaintext has a particular difference. The exclusive-or of such pairs is called a

differential and certain differentials have certain probabilities associated to them, depending on what the key is. By analysing the probabilities of the differentials computed in a chosen plaintext attack one can hope to reveal the underlying structure of the key.

- **Linear Cryptanalysis:** Even though a good block cipher should contain non-linear components the idea behind linear cryptanalysis is to approximate the behaviour of the non-linear components with linear functions. Again the goal is to use a probabilistic analysis to determine information about the key.

Surprisingly these two methods are quite successful against some ciphers. But they do not appear that successful against DES or Rijndael, two of the most important block ciphers in use today.

Since DES and Rijndael are likely to be the most important block ciphers in use for the next few years we shall study them in some detail. This is also important since they both show general design principles in their use of substitutions and permutations. Recall that the historical ciphers made use of such operations, so we see that not much has changed. Now, however, the substitutions and permutations used are far more intricate. On their own they do not produce security, but when used over a number of rounds one can obtain enough security for our applications.

We end this section by discussing which is best, a block cipher or a stream cipher? Alas there is no correct answer to this question. Both have their uses and different properties. Here are just a few general points.

- Block ciphers are more general, and we shall see that one can easily turn a block cipher into a stream cipher.
- Stream ciphers generally have a more mathematical structure. This either makes them easier to break or easier to study to convince oneself that they are secure.
- Stream ciphers are generally not suitable for software, since they usually encrypt one bit at a time. However, stream ciphers are highly efficient in hardware.
- Block ciphers are suitable for both hardware and software, but are not as fast in hardware as stream ciphers.
- Hardware is always faster than software, but this performance improvement comes at the cost of less flexibility.

2. Feistel Ciphers and DES

The DES cipher is a variant of the basic Feistel cipher described in Fig. 2, named after H. Feistel who worked at IBM and performed some of the earliest non-military research on encryption algorithms. The interesting property of a Feistel cipher is that the round function is invertible regardless of the choice of the function in the box marked F. To see this notice that each encryption round is given by

$$\begin{aligned} L_i &= R_{i-1}, \\ R_i &= L_{i-1} \oplus F(K_i, R_{i-1}). \end{aligned}$$

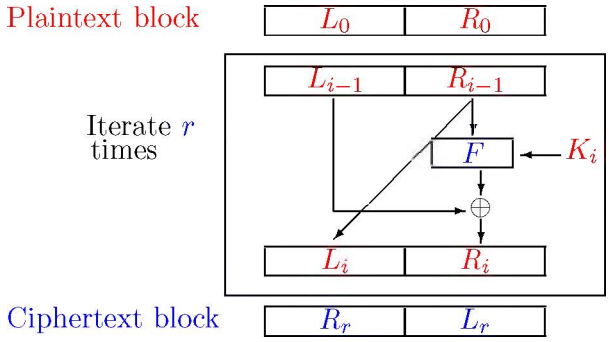
Hence, the decryption can be performed via

$$\begin{aligned} R_{i-1} &= L_i, \\ L_{i-1} &= R_i \oplus F(K_i, L_i). \end{aligned}$$

This means that in a Feistel cipher we have simplified the design somewhat, since

- we can choose any function for the function F, and we will still obtain an encryption function which can be inverted using the secret key,

FIGURE 2. Basic operation of a Feistel cipher



- the same code/circuitry can be used for the encryption and decryption functions. We only need to use the round keys in the reverse order for decryption.

Of course to obtain a secure cipher we still need to take care with

- how the round keys are generated,
- how many rounds to take,
- how the function F is defined.

Work on DES was started in the early 1970s by a team in IBM which included Feistel. It was originally based on an earlier cipher of IBM's called Lucifer, but some of the design was known to have been amended by the National Security Agency, NSA. For many years this led the conspiracy theorists to believe that the NSA had placed a trapdoor into the design of the function F. However, it is now widely accepted that the modifications made by the NSA were done to make the cipher more secure. In particular, the changes made by the NSA made the cipher resistant to differential cryptanalysis, a technique that was not discovered in the open research community until the 1980s.

DES is also known as the Data Encryption Algorithm DEA in documents produced by the American National Standards Institute, ANSI. The International Standards Organisation ISO refers to DES by the name DEA-1. It has been a world-wide standard for well over twenty years and stands as the first publicly available algorithm to have an 'official status'. It therefore marks an important step on the road from cryptography being a purely military area to being a tool for the masses.

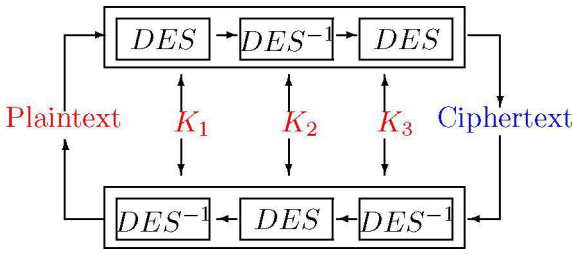
The basic properties of the DES cipher are that it is a variant of the Feistel cipher design with

- the number of rounds r is 16,
- the block length n is 64 bits,
- the key length is 56 bits,
- the round keys $K_1, \ldots, K_{16}$ are each 48 bits.

Note that a key length of 56 bits is insufficient for many modern applications, hence often one uses DES by using three keys and three iterations of the main cipher. Such a version is called Triple DES or 3DES, see Fig. 3. In 3DES the key length is equal to 168. There is another way of using DES three times, but using two keys instead of three giving rise to a key length of 112. In this two-key version of 3DES one uses the 3DES basic structure but with the first and third key being equal. However, two-key 3DES is not as secure as one might inititally think.

2.1. Overview of DES Operation. Basically DES is a Feistel cipher with 16 rounds, as depicted in Fig. 4, except that before and after the main Feistel iteration a permutation is performed. Notice how the two blocks are swapped around before being passed through the final inverse permutation. This permutation appears to produce no change to the security, and people have often wondered why it is there. One answer given by one of the original team members was that this permutation was there to make the original implementation easier to fit on the circuit board.

FIGURE 3. Triple DES

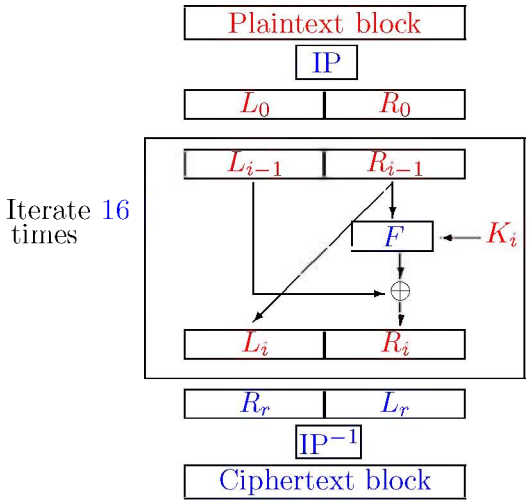


In summary the DES cipher operates on 64 bits of plaintext in the following manner:

- Perform an initial permutation.
- Split the blocks into left and right half.
- Perform 16 rounds of identical operations.
- Join the half blocks back together.
- Perform a final permutation.

The final permutation is the inverse of the initial permutation, this allows the same hardware/software to be used for encryption and decryption. The key schedule provides 16 round keys of 48 bits in length by selecting 48 bits from the 56-bit main key.

FIGURE 4. DES as a Feistel cipher



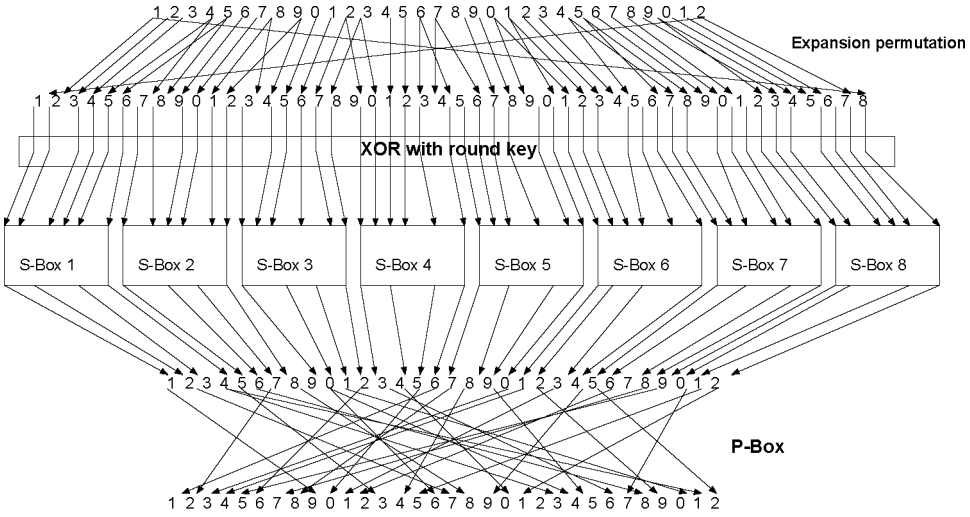
We shall now describe the operation of the function F. In each DES round this consists of the following six stages:

- **Expansion Permutation:** The right half of 32 bits is expanded and permuted to 48 bits. This helps the diffusion of any relationship of input bits to output bits. The expansion permutation (which is different from the initial permutation) has been chosen so that one bit of input affects two substitutions in the output, via the S-Boxes below. This helps spread dependencies and creates an avalanche effect (a small difference between two plaintexts will produce a very large difference in the corresponding ciphertexts).
- **Round Key Addition:** The 48-bit output from the expansion permutation is XORed with the round key, which is also 48 bits in length. Note, this is the only place where the round key is used in the algorithm.
- **Splitting:** The resulting 48-bit value is split into eight lots of six-bit values.

- **S-Box:** Each six-bit value is passed into one of eight different S-Boxes (Substitution Box) to produce a four-bit result. The S-Boxes represent the non-linear component in the DES algorithm and their design is a major contributor to the algorithms security. Each S-Box is a look-up table of four rows and sixteen columns. The six input bits specify which row and column to use. Bits 1 and 6 generate the row number, whilst bits $2, 3, 4$ and 5 specify the column number. The output of each S-Box is the value held in that element in the table.
- **P-Box:** We now have eight lots of four-bit outputs which are then combined into a 32-bit value and permuted to form the output of the function F.

The overall structure of DES is explained in Fig. 5.

FIGURE 5. Structure of the DES function F



We now give details of each of the steps which we have not yet fully defined.

2.1.1. *Initial Permutation, IP:.* The DES initial permutation is defined in the following table. Here the 58 in the first position means that the first bit of the output from the IP is the 58th bit of the input, and so on.

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

The inverse permutation is given in a similar manner by the following table.

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

2.1.2. *Expansion Permutation, E:.* The expansion permutation is given in the following table. Each row corresponds to the bits which are input into the corresponding S-Box at the next stage. Notice how the bits which select a row of one S-Box (the first and last bit on each row) are also used to select the column of another S-Box.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

2.1.3. *S-Box:* The details of the eight DES S-Boxes are given in the Fig. 6. Recall that each box consists of a table with four rows and sixteen columns.

2.1.4. *The P-Box Permutation, P:.* The P-Box permutation takes the eight lots of four-bit nibbles, output of the S-Boxes, and produces a 32-bit permutation of these values as given by the following table.

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

2.2. DES Key Schedule. The DES key schedule takes the 56-bit key, which is actually input as a bitstring of 64 bits comprising of the key and eight parity bits, for error detection. These parity bits are in bit positions $8, 16, \ldots, 64$ and ensure that each byte of the key contains an odd number of bits.

We first permute the bits of the key according to the following permutation (which takes a 64-bit input and produces a 56-bit output, hence discarding the parity bits).

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

FIGURE 6. DES S-Boxes

S-Box 1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S-Box 2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S-Box 3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S-Box 4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S-Box 5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S-Box 6

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S-Box 7

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S-Box 8

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

The output of this permutation, called PC-1 in the literature, is divided into a 28-bit left half C_0 and a 28-bit right half D_0. Now for each round we compute

$$C_i = C_{i-1} \lll p_i,$$
$$D_i = D_{i-1} \lll p_i,$$

where $x \lll p_i$ means perform a cyclic shift on x to the left by p_i positions. If the round number i is $1, 2, 9$ or 16 then we shift left by one position, otherwise we shift left by two positions.

Finally the two portions C_i and D_i are joined back together and are subject to another permutation, called PC-2, to produce the final 48-bit round key. The permutation PC-2 is described below.

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

3. Rijndael

The AES winner was decided in fall 2000 to be the Rijndael algorithm designed by Daemen and Rijmen. Rijndael is a block cipher which does not rely on the basic design of the Feistel cipher. However, Rijndael does have a number of similarities with DES. It uses a repeated number of rounds to obtain security and each round consists of substitutions and permutations, plus a key addition phase. Rijndael in addition has a strong mathematical structure, as most of its operations are based on arithmetic in the field $\mathbb{F}_{2^8}$. However, unlike DES the encryption and decryption operations are distinct.

Recall that elements of $\mathbb{F}_{2^8}$ are stored as bit vectors (or bytes) representing binary polynomials. For example the byte given by $0x83$ in hexadecimal, gives the bit pattern

$$1, 0, 0, 0, 0, 0, 1, 1$$

since

$$0x83 = 8 \cdot 16 + 3 = 131$$

in decimal. One can obtain the bit pattern directly by noticing that 8 in binary is $1, 0, 0, 0$ and 3 in 4-bit binary is $0, 0, 1, 1$ and one simply concatenates these two bit strings together. The bit pattern itself then corresponds to the binary polynomial

$$x^7 + x + 1.$$

So we say that the hexadecimal number $0x83$ represents the binary polynomial

$$x^7 + x + 1.$$

Arithmetic in $\mathbb{F}_{2^8}$ is performed using polynomial arithmetic modulo the irreducible polynomial

$$m(x) = x^8 + x^4 + x^3 + x + 1.$$

Rijndael identifies 32-bit words with polynomials in $\mathbb{F}_{2^8}[X]$ of degree less than four. This is done in a big-endian format, in that the smallest index corresponds to the least important coefficient. Hence, the word

$$a_0 \| a_1 \| a_2 \| a_3$$

will correspond to the polynomial

$$a_3 X^3 + a_2 X^2 + a_1 X + a_0.$$

Arithmetic is performed on polynomials in $\mathbb{F}_{2^8}[X]$ modulo the reducible polynomial

$$M(X) = X^4 + 1.$$

Hence, arithmetic is done on these polynomials in a ring rather than a field, since $M(X)$ is reducible.

Rijndael is a parametrized algorithm in that it can operate on block sizes of 128, 192 or 256 bits, it can also accept keys of size 128, 192 or 256 bits. For each combination of block and key size a different number of rounds is specified. To make our discussion simpler we shall consider

the simpler, and probably more used, variant which uses a block size of 128 bits and a key size of 128 bits, in which case 10 rounds are specified. From now on our discussion is only of this simpler version.

Rijndael operates on an internal four-by-four matrix of bytes, called the state matrix

$$S = \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix},$$

which is usually held as a vector of four 32-bit words, each word representing a column. Each round key is also held as a four-by-four matrix

$$K_i = \begin{pmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{pmatrix}.$$

3.1. Rijndael Operations. The Rijndael round function operates using a set of four operations which we shall first describe.

3.1.1. *SubBytes:* There are two types of S-Boxes used in Rijndael: One for the encryption rounds and one for the decryption rounds, each one being the inverse of the other. We shall describe the encryption S-Box, the decryption one will follow immediately. The S-Boxes of DES were chosen by searching through a large space of possible S-Boxes, so as to avoid attacks such as differential cryptanalysis. The S-Box of Rijndael is chosen to have a simple mathematical structure, which allows one to formally argue how resilient the cipher is from differential and linear cryptanalysis. Not only does this mathematical structure help protect against differential cryptanalysis, but it also convinces users that it has not been engineered with some hidden trapdoor.

Each byte $s = [s_7, \ldots, s_0]$ of the Rijndael state matrix is taken in turn and considered as an element of $\mathbb{F}_{2^8}$. The S-Box can be mathematically described in two steps:

(1) The multiplicative inverse in $\mathbb{F}_{2^8}$ of s is computed to produce a new byte $x = [x_7, \ldots, x_0]$. For the element $[0, \ldots, 0]$ which has no multiplicative inverse one uses the convention that this is mapped to zero.
(2) The bit-vector x is then mapped, via the following affine $\mathbb{F}_2$ transformation, to the bit-vector y:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

The new byte is given by y. The decryption S-Box is obtained by first inverting the affine transformation and then taking the multiplicative inverse. These byte substitutions can either be implemented using table look-up or by implementing circuits, or code, which implement the inverse operation in $\mathbb{F}_{2^8}$ and the affine transformation.

3.1.2. *ShiftRows:* The ShiftRows operation in Rijndael performs a cyclic shift on the state matrix. Each row is shifted by different offsets. For the version of Rijndael we are considering this

is given by

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} \longmapsto \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{pmatrix}.$$

The inverse of the ShiftRows operation is simply a similar shift but in the opposite direction. The ShiftRows operation ensures that the columns of the state matrix 'interact' with each other over a number of rounds.

3.1.3. *MixColumns:* The MixColumns operation ensures that the rows in the state matrix 'interact' with each other over a number of rounds; combined with the ShiftRows operation it ensures each byte of the output state depends on each byte of the input state.

We consider each column of the state in turn and consider it as a polynomial of degree less than four with coefficients in $\mathbb{F}_{2^8}$. The new column $[b_0, b_1, b_2, b_3]$ is produced by taking this polynomial

$$a(X) = a_0 + a_1X + a_2X^2 + a_3X^3$$

and multiplying it by the polynomial

$$c(X) = 0x02 + 0x01 \cdot X + 0x01 \cdot X^2 + 0x03 \cdot X^3$$

modulo

$$M(X) = X^4 + 1.$$

This operation is conveniently represented by the following matrix operation in $\mathbb{F}_{2^8}$,

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}.$$

In $\mathbb{F}_{2^8}$ the above matrix is invertible, hence the inverse of the MixColumns operation can also be implemented using a matrix multiplication such as that above.

3.1.4. *AddRoundKey:* The round key addition is particularly simple. One takes the state matrix and XORs it, byte by byte, with the round key matrix. The inverse of this operation is clearly the same operation.

3.2. Round Structure. The Rijndael algorithm can now be described using the pseudo-code in Algorithm 8.1. The message block to encrypt is assumed to be entered into the state matrix S, the output encrypted block is also given by the state matrix S. Notice that the final round does not perform a MixColumns operation. The corresponding decryption operation is described in Algorithm 8.2.

Algorithm 8.1: Rijndael Encryption Outline

```
AddRoundKey(S, K_0)
for i = 1 to 9 do
    SubBytes(S)
    ShiftRows(S)
    MixColumns(S)
    AddRoundKey(S, K_i)
end
SubBytes(S)
ShiftRows(S)
AddRoundKey(S, K_10)
```

Algorithm 8.2: Rijndael Decryption Outline

AddRoundKey(S, K_{10})
InverseShiftRows(S)
InverseSubBytes(S)
for $i = 9$ **downto** 1 **do**
  AddRoundKey(S, K_i)
  InverseMixColumns(S)
  InverseShiftRows(S)
  InverseSubBytes(S)
end
AddRoundKey(S, K_0)

3.3. Key Schedule. The only thing left to describe is how Rijndael computes the round keys from the main key. Recall that the main key is 128 bits long, and we need to produce 11 round keys $K_0, \ldots, K_{11}$ all of which consist of four 32-bit words. Each word corresponding to a column of a matrix as described above. The key schedule makes use of a round constant which we shall denote by

$$RC_i = x^i \pmod{x^8 + x^4 + x^3 + x + 1}.$$

We label the round keys as $(W_{4i}, W_{4i+1}, W_{4i+2}, W_{4i+3})$ where i is the round. The initial main key is first divided into four 32-bit words (k_0, k_1, k_2, k_3). The round keys are then computed as in Algorithm 8.3, where RotBytes is the function which rotates a word to the left by a single byte, and SubBytes applies the Rijndael encryption S-Box to every byte in a word.

Algorithm 8.3: Rijndael Key Schedule

$W_0 = K_0, W_1 = K_1, W_2 = K_2, W_3 = K_3$
for $i = 1$ **to** 10 **do**
  $T =$RotBytes(W_{4i-1})
  $T =$SubBytes(T)
  $T = T \oplus RC_i$
  $W_{4i} = W_{4i-4} \oplus T$
  $W_{4i+1} = W_{4i-3} \oplus W_{4i}$
  $W_{4i+2} = W_{4i-2} \oplus W_{4i+1}$
  $W_{4i+3} = W_{4i-1} \oplus W_{4i+2}$
end

4. Modes of Operation

A block cipher like DES or Rijndael can be used in a variety of ways to encrypt a data string. Soon after DES was standardized another US Federal standard appeared giving four *recommended* ways of using DES for data encryption. These modes of operation have since been standardized internationally and can be used with any block cipher. The four modes are

- **ECB Mode:** This is simple to use, but suffers from possible deletion and insertion attacks. A one-bit error in ciphertext gives one whole block error in the decrypted plaintext.
- **CBC Mode:** This is the best mode to use as a block cipher since it helps protect against deletion and insertion attacks. In this mode a one-bit error in the ciphertext gives not only a one-block error in the corresponding plaintext block but also a one-bit error in the next decrypted plaintext block.

- **OFB Mode:** This mode turns a block cipher into a stream cipher. It has the property that a one-bit error in ciphertext gives a one-bit error in the decrypted plaintext.
- **CFB Mode:** This mode also turns a block cipher into a stream cipher. A single bit error in the ciphertext affects both this block and the next, just as in CBC mode.

Over the years various other modes of operation have been presented, probably the most popular of the more modern modes is

- **CTR Mode:** This also turns the block cipher into a stream cipher, but in enables blocks to be processed in parallel, thus providing performance advantages when parallel processing is available.

We shall now describe each of these five modes of operation in detail.

4.1. ECB Mode. Electronic Code Book Mode, or ECB Mode, is the simplest way to use a block cipher. The data to be encrypted m is divided into blocks of n bits:

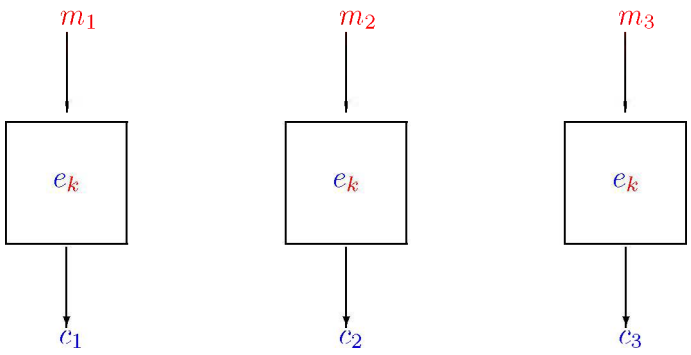
$$m_1, m_2, \ldots, m_q$$

with the last block padded if needed. The ciphertext blocks $c_1, \ldots, c_q$ are then defined as follows

$$c_i = e_k(m_i),$$

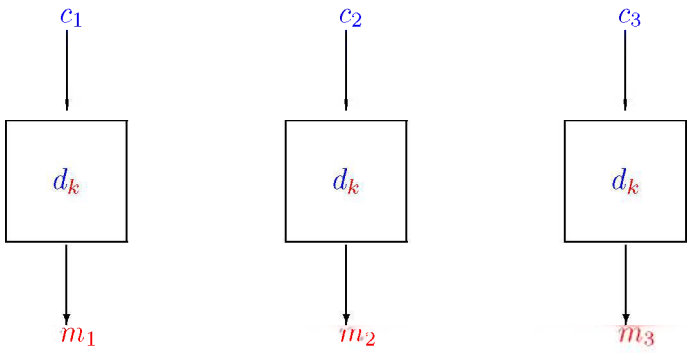
as described in Fig. 7. Decipherment is simply the reverse operation as explained in Fig. 8.

FIGURE 7. ECB encipherment



ECB Mode has a number of problems: the first is due to the property that if $m_i = m_j$ then we have $c_i = c_j$, i.e. the same input block always generates the same output block. This is a problem since stereotyped beginnings and ends of messages are common. The second problem comes because

FIGURE 8. ECB decipherment



we could simply delete blocks from the message and no one would know. Thirdly we could replay known blocks from other messages. By extracting ciphertext corresponding to a known piece of plaintext we can then amend other transactions to contain this known block of text.

To see all these problems suppose our block cipher is rather simple and encrypts each English word as a block. Suppose we obtained the encryption of the sentences

Pay Alice one hundred pounds,
Don't pay Bob two hundred pounds,

which encrypted were

the horse has four legs,
stop the pony hasn't four legs.

We can now make the recipient pay Alice two hundred pounds by sending her the message

the horse hasn't four legs,

in other words we have replaced a block from one message by a block from another message. Or we could stop the recipient paying Alice one hundred pounds by inserting the encryption stop of don't onto the front of the original message to Alice. Or we can make the recipient pay Bob two hundred pounds by deleting the first block of the message sent to him.

These threats can be countered by adding checksums over a number of plaintext blocks, or by using a mode of operation which adds some 'context' to each ciphertext block.

4.2. CBC Mode. One way of countering the problems with ECB Mode is to chain the cipher, and in this way add context to each ciphertext block. The easiest way of doing this is to use Cipher Block Chaining Mode, or CBC Mode.

Again, the plaintext must first be divided into a series of blocks

$$m_1, \ldots, m_q,$$

and as before the final block may need padding to make the plaintext length a multiple of the block length. Encryption is then performed via the equations

$$\begin{aligned} c_1 &= e_k(m_1 \oplus IV), \\ c_i &= e_k(m_i \oplus c_{i-1}) \text{ for } i > 1, \end{aligned}$$

see also Fig. 9.

Notice that we require an additional initial value IV to be passed to the encryption function, which can be used to make sure that two encryptions of the same plaintext produce different ciphertexts. In some situations one therefore uses a random IV with every message, usually when the same key will be used to encrypt a number of messages. In other situations, mainly when the key to the block cipher is only going to be used once, one chooses a fixed IV, for example the all zero string. In the case where a random IV is used, it is not necessary for the IV to be kept secret and it is usually transmitted in the clear from the encryptor to the decryptor as part of the message. The distinction between the reasons for using a fixed or random value for IV is expanded upon further in Chapter 21.

Decryption also requires the IV and is performed via the equations,

$$\begin{aligned} m_1 &= d_k(c_1) \oplus IV, \\ m_i &= d_k(c_i) \oplus c_{i-1} \text{ for } i > 1, \end{aligned}$$

see Fig. 10.

With ECB Mode a single bit error in transmission of the ciphertext will result in a whole block being decrypted wrongly, whilst in CBC Mode we see that not only will we decrypt a block incorrectly but the error will also affect a single bit of the next block.

FIGURE 9. CBC encipherment

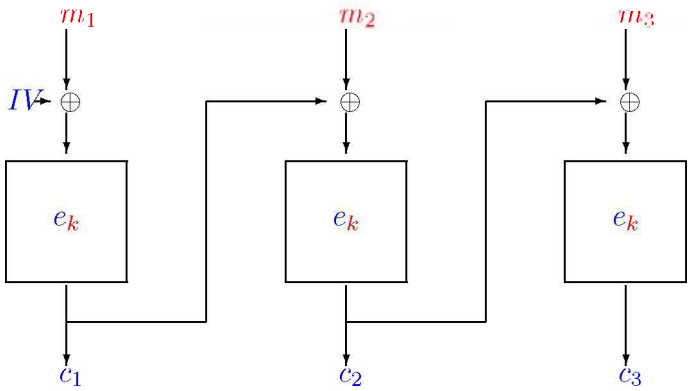
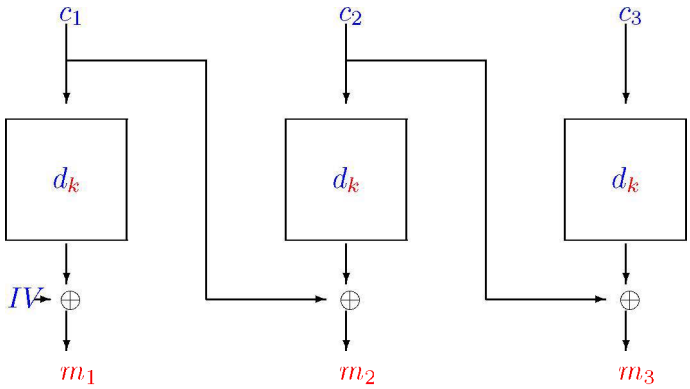


FIGURE 10. CBC decipherment



4.3. OFB Mode. Output Feedback Mode, or OFB Mode enables a block cipher to be used as a stream cipher. We need to choose a variable j $(1 \leq j \leq n)$ which will denote the number of bits output by the keystream generator on each iteration. We use the block cipher to create the keystream, j bits at a time. It is however usually recommended to take $j = n$ as that makes the expected cycle length of the keystream generator larger.

Again we divide plaintext into a series of blocks, but this time each block is j-bits, rather than n-bits long:

$$m_1, \ldots, m_q.$$

Encryption is performed as follows, see Fig. 11 for a graphical representation. First we set $X_1 = IV$, then for $i = 1, 2, \ldots, q$, we perform the following steps,

$$\begin{aligned} Y_i &= e_k(X_i), \\ E_i &= j \text{ leftmost bits of } Y_i, \\ c_i &= m_i \oplus E_i, \\ X_{i+1} &= Y_i. \end{aligned}$$

Decipherment in OFB Mode is performed in a similar manner as described in Fig. 12.

4.4. CFB Mode. The next mode we consider is called Cipher FeedBack Mode, or CFB Mode. This is very similar to OFB Mode in that we use the block cipher to produce a stream cipher. Recall

FIGURE 11. OFB encipherment

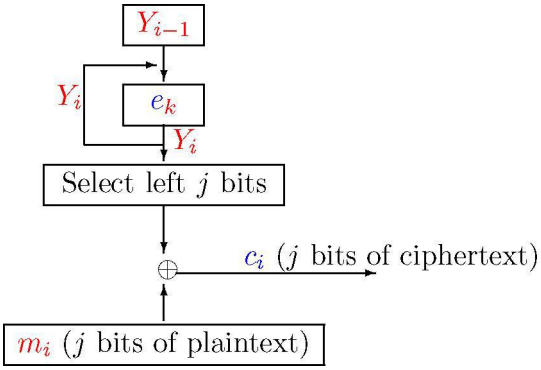
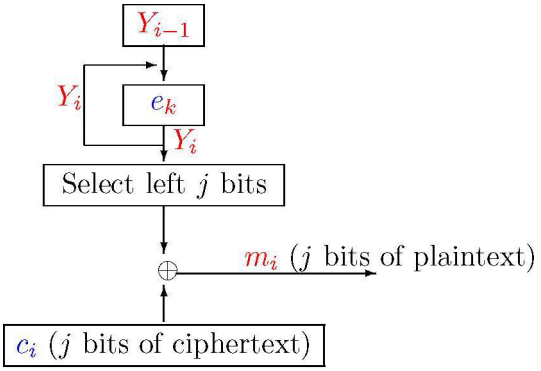


FIGURE 12. OFB decipherment



that in OFB Mode the keystream was generated by encrypting the IV and then iteratively encrypting the output from the previous encryption. In CFB Mode the keystream output is produced by the encryption of the ciphertext, as in Fig. 13, by the following steps,

$$\begin{aligned} Y_0 &= IV, \\ Z_i &= e_k(Y_{i-1}), \\ E_i &= j \text{ leftmost bits of } Z_i, \\ Y_i &= m_i \oplus E_i. \end{aligned}$$

We do not present the decryption steps, but leave these as an exercise for the reader.

4.5. CTR Mode. The next mode we consider is called Counter Mode, or CTR Mode. This combines many of the advantages of ECB Mode, but with none of the disadvantages. We first select a public IV, or counter, which is chosen differently for each message encrypted under the fixed key k. Then encryption proceeds for the ith block, by encrypting the value of $IV + i$ and then xor'ing this with the message block. In other words we have

$$c_i = m_i \oplus e_k(IV + i).$$

This is explained pictorially in Figure 14

CTR Mode has a number of interesting properties. Firstly since each block can be encrypted independently, much like in ECB Mode, we can process each block at the same time. Compare this to CBC Mode, OFB Mode or CFB Mode where we cannot start encrypting the second block until the first block has been encrypted. This means that encryption, and decryption, can be performed in parallel. However, unlike ECB Mode two equal blocks will not encrypt to the same ciphertext

FIGURE 13. CFB encipherment

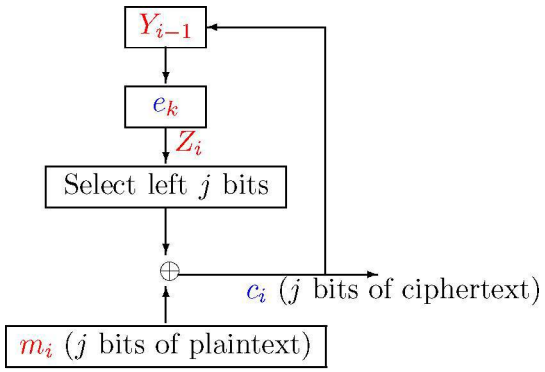
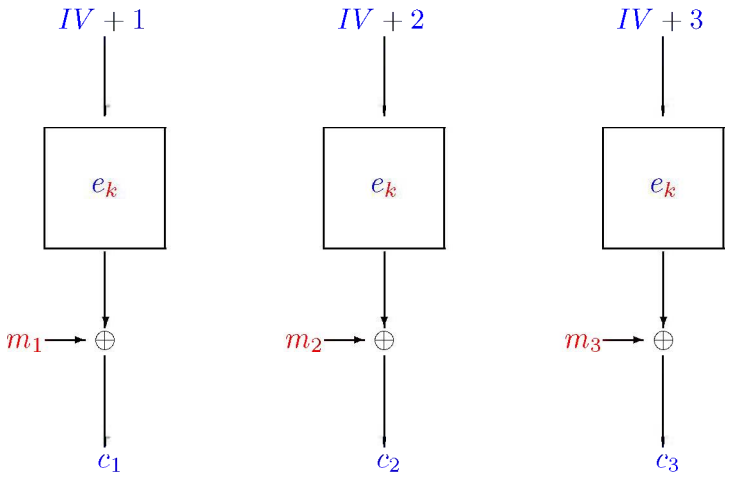


FIGURE 14. CTR encipherment



value. This is because each plaintext block is encrypted using a different input to the encryption function, in some sense we are using the block cipher encryption of the different inputs to produce a stream cipher. Also unlike ECB Mode each ciphertext block corresponds to a precise position within the ciphertext, as its position information is needed to be able to decrypt it successfully.

Chapter Summary

- The most popular block cipher is DES, which is itself based on a general design called a Feistel cipher.
- A comparatively recent block cipher is the AES cipher, called Rijndael.
- Both DES and Rijndael obtain their security by repeated application of simple rounds consisting of substitution, permutation and key addition.
- To use a block cipher one needs to also specify a mode of operation. The simplest mode is ECB mode, which has a number of problems associated with it. Hence, it is common to use a more advanced mode such as CBC or CTR mode.

- Some block cipher modes, such as CFB, OFB and CTR modes, allow the block cipher to be used in a stream cipher.

Further Reading

The Rijndael algorithm, the AES process and a detailed discussion of attacks on block ciphers and Rijndael in particular can be found in the book by Daemen and Rijmen. Stinson's book is the best book to explain differential cryptanalysis for students.

J. Daemen and V. Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard.* Springer-Verlag, 2002.

D. Stinson. *Cryptography Theory and Practice.* CRC Press, 1995.

CHAPTER 9

Symmetric Key Distribution

Chapter Goals

- To understand the problems associated with managing and distributing secret keys.
- To learn about key distribution techniques based on symmetric key based protocols.
- To introduce the formal analysis of protocols.

1. Key Management

To be able to use symmetric encryption algorithms such as DES or Rijndael we need a way for the two communicating parties to share the secret key. In this first section we discuss some issues related to how keys are managed, in particular

- key distribution,
- key selection,
- key lifetime,

But before we continue we need to distinguish between different types of keys. The following terminology will be used throughout this chapter and beyond:

- **Static (or long-term) Keys:** These are keys which are to be in use for a long time period. The exact definition of long will depend on the application, but this could mean from a few hours to a few years. The compromise of a static key is usually considered to be a major problem, with potentially catastrophic consequences.
- **Ephemeral, or Session (or short-term) Keys:** These are keys which have a short life-time, maybe a few seconds or a day. They are usually used to provide confidentiality for the given time period. The compromise of a session key should only result in the compromise of that session's secrecy and it should not affect the long-term security of the system.

1.1. Key Distribution. Key distribution is one of the fundamental problems with cryptography. There are a number of solutions to this problem; which one of these one chooses depends on the overall situation.

- **Physical Distribution:** Using trusted couriers or armed guards, keys can be distributed using traditional physical means. Until the 1970s this was in effect the only secure way of distributing keys at system setup. It has a large number of physical problems associated with it, especially scalability, but the main drawback is that security no longer rests with the key but with the courier. If we can bribe, kidnap or kill the courier then we have broken the system.
- **Distribution Using Symmetric Key Protocols:** Once some secret keys have been distributed between a number of users and a trusted central authority, we can use the trusted authority to help generate keys for any pair of users as the need arises. Protocols to perform this task will be discussed in this chapter. They are usually very efficient but

have some drawbacks. In particular they usually assume that both the trusted authority and the two users who wish to agree on a key are both on-line. They also still require a physical means to set the initial keys up.

- **Distribution Using Public Key Protocols:** Using public key cryptography, two parties, who have never met or who do not trust any one single authority, can produce a shared secret key. This can be done in an on-line manner, using a key exchange protocol. Indeed this is the most common application of public key techniques for encryption. Rather than encrypting large amounts of data by public key techniques we agree a key by public key techniques and then use a symmetric cipher to actually do the encryption.

To understand the scale of the problem, if our system is to cope with n separate users, and each user may want to communicate securely with any other user, then we require

$$\frac{n(n-1)}{2}$$

separate secret keys. This soon produces huge key management problems; a small university with around 10 000 students would need to have around fifty million separate secret keys.

With a large number of keys in existence one finds a large number of problems. For example what happens when your key is compromised? In other words someone else has found your key. What can you do about it? What can they do? Hence, a large number of keys produces a large key management problem.

One solution is for each user to hold only one key with which it communicates with a central authority, hence a system with n users will only require n keys. When two users wish to communicate they generate a secret key which is only to be used for that message, a so-called session key. This session key can be generated with the help of the central authority using one of the protocols that appear later in this chapter.

1.2. Key Selection. The keys which one uses should be truly random, since otherwise an attacker may be able to determine information simply by knowing the more likely keys and the more likely messages, as we saw in a toy example in Chapter 5. All keys should be equally likely and really need to be generated using a true random number generator, however such a good source of entropy is hard to find.

Whilst a truly random key will be very strong, it is hard for a human to remember. Hence, many systems use a password or pass phrase to generate a secret key. But now one needs to worry even more about brute force attacks. As one can see from the following table, a typical PIN-like password of a number between 0 and 9999 is easy to mount a brute force attack against, but even using eight printable characters does not push us to the 2^{80} possibilities that we would like to ensure security.

Key size	Decimal digits	Printable characters
4	$10^4 \approx 2^{13}$	$10^7 \approx 2^{23}$
8	$10^8 \approx 2^{26}$	$10^{15} \approx 2^{50}$

One solution may be to use long pass phrases of 20–30 characters, but these are likely to lack sufficient entropy since we have already seen that natural language is not very random.

Short passwords based on names or words are a common problem in many large organizations. This is why a number of organizations now have automatic checking that passwords meet certain criteria such as

- at least one lower case letter,
- at least one upper case letter,
- at least one numeric character,
- at least one non-alpha-numeric character,
- at least eight characters in length.

But such rules, even though they eliminate the chance of a dictionary attack, still reduce the number of possible passwords from what they would be if they were chosen uniformly at random from all choices of eight printable characters.

1.3. Key Lifetime. One issue one needs to consider when generating and storing keys is the key lifetime. A general rule is that the longer the key is in use the more vulnerable it will be and the more valuable it will be to an attacker. We have already touched on this when mentioning the use of session keys. However, it is important to destroy keys properly after use. Relying on an operating system to delete a file by typing `del/rm` does not mean that an attacker cannot recover the file contents by examining the hard disk. Usually deleting a file does not destroy the file contents, it only signals that the file's location is now available for overwriting with new data. A similar problem occurs when deleting memory in an application.

1.4. Secret Sharing. As we have mentioned already the main problem is one of managing the secure distribution of keys. Even a system which uses a trusted central authority needs some way of getting the keys shared between the centre and each user out to the user.

One possible solution is key splitting (more formally called *secret sharing*) where we divide the key into a number of shares

$$K = k_1 \oplus k_2 \oplus \cdots \oplus k_r.$$

Each share is then distributed via separate routes. The beauty of this is that an attacker needs to attack all the routes so as to obtain the key. On the other hand attacking one route will stop the legitimate user from recovering the key.

We will discuss secret sharing in more detail in Chapter 23.

2. Secret Key Distribution

Recall, if we have n users each of whom wish to communicate securely with each other then we would require

$$\frac{n(n-1)}{2}$$

separate long-term key pairs. As remarked earlier this leads to huge key management problems and issues related to the distribution of the keys. We have already mentioned that it is better to use session keys and few long-term keys, but we have not explained how one deploys the session keys.

To solve this problem the community developed a number of protocols which make use of symmetric key cryptography to distribute secret session keys, some of which we shall describe in this section. Later on we shall look at public key techniques for this problem, which are often more elegant.

2.1. Notation. We first need to set up some notation to describe the protocols. Firstly we set up the names of the parties and quantities involved.

- **Parties/Principals:** A, B, S.
  Assume the two parties who wish to agree a secret are A and B, for Alice and Bob. We assume that they will use a trusted third party, or TTP, which we shall denote by S.
- **Shared Secret Keys:** K_{ab}, K_{bs}, K_{as}.
  K_{ab} will denote a secret key known only to A and B.
- **Nonces:** N_a, N_b.
  Nonces are numbers used only once, they should be random. The quantity N_a will denote a nonce originally produced by the principal A. Note, other notations for nonces are possible and we will introduce them as the need arises.

- **Timestamps:** T_a, T_b, T_s.
  The quantity T_a is a timestamp produced by A. When timestamps are used we assume that the parties try to keep their clocks in synchronization using some other protocol.

The statement

$$A \longrightarrow B : M, A, B, \{N_a, M, A, B\}_{K_{as}},$$

means A sends to B the message to the right of the colon. The message consists of

- a nonce M,
- A the name of party A,
- B the name of party B,
- a message $\{N_a, M, A, B\}$ encrypted under the key K_{as} which A shares with S. Hence, the recipient B is unable to read the encrypted part of this message.

Before presenting our first protocol we need to decide the goals of key agreement and key transport, and what position the parties start from. We assume all parties, A and B say, only share secret keys, K_{as} and K_{bs} with the trusted third party S. They want to agree/transport a session key K_{ab} for a communication between themselves.

We also need to decide what capabilities an attacker has. As always we assume the worst possible situation in which an attacker can intercept any message flow over the network. She can then stop a message, alter it or change its destination. An attacker is also able to distribute her own messages over the network. With such a high-powered attacker it is often assumed that the attacker *is* the network.

This new session key should be fresh, i.e. it has not been used by any other party before and has been recently created. The freshness property will stop attacks whereby the adversary replays messages so as to use an old key again. Freshness also can be useful in deducing that the party with which you are communicating is still alive.

2.2. Wide-Mouth Frog Protocol. Our first protocol is the Wide-Mouth Frog protocol, which is a simple protocol invented by Burrows. The protocol transfers a key K_{ab} from A to B via S, it uses only two messages but has a number of drawbacks. In particular it requires the use of synchronized clocks, which can cause a problem in implementations. In addition the protocol assumes that A chooses the session key K_{ab} and then transports this key over to user B. This implies that user A is trusted by user B to be competent in making and keeping keys secret. This is a very strong assumption and the main reason that this protocol is not used much in real life. However, it is very simple and gives a good example of how to analyse a protocol formally, which we shall come to later in this chapter.

The protocol proceeds in the following steps, as illustrated in Fig. 1,

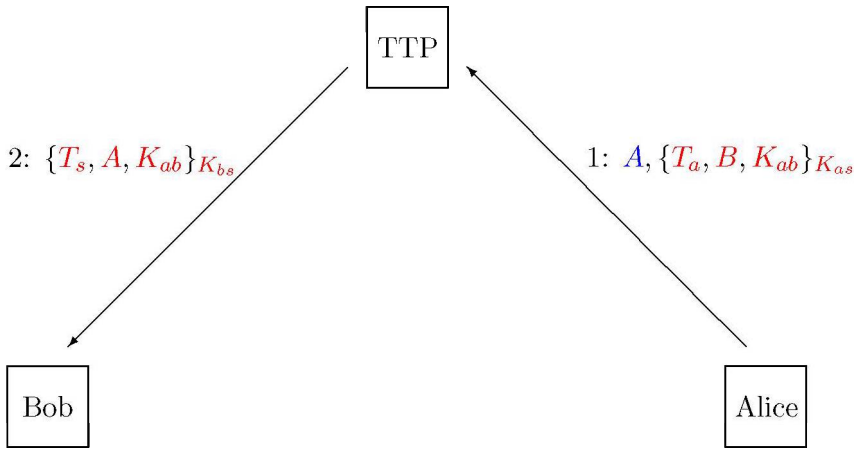
$$A \longrightarrow S : A, \{T_a, B, K_{ab}\}_{K_{as}},$$
$$S \longrightarrow B : \{T_s, A, K_{ab}\}_{K_{bs}}.$$

On obtaining the first message the trusted third party S decrypts the last part of the message and checks that the timestamp is recent. This decrypted message tells S he should forward the key to the party called B. If the timestamp is verified to be recent, S encrypts the key along with his timestamp and passes this encryption onto B. On obtaining this message B decrypts the message received and checks the time stamp is recent, then he can recover both the key K_{ab} and the name A of the person who wants to send data to him using this key.

The checks on the timestamps mean the session key should be recent, in that it left user A a short time ago. However, user A could have generated this key years ago and stored it on his hard disk, in which time Eve broke in and took a copy of this key.

We already said that this protocol requires that all parties need to keep synchronized clocks. However, this is not such a big problem since S checks or generates all the timestamps used in the

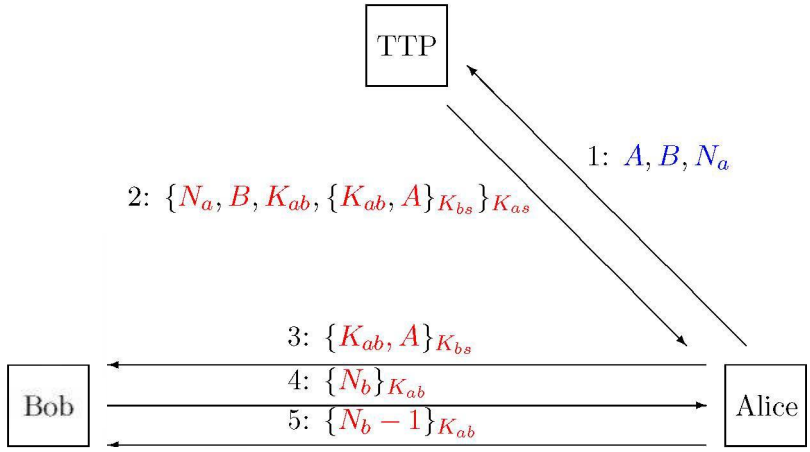
FIGURE 1. Wide-Mouth Frog protocol



protocol. Hence, each party only needs to record the difference between its clock and the clock owned by S. Clocks are then updated if a clock drift occurs which causes the protocol to fail.

This protocol is really too simple; much of the simplicity comes by assuming synchronized clocks and by assuming party A can be trusted with creating session keys.

FIGURE 2. Needham–Schroeder protocol



2.3. Needham–Schroeder Protocol. We shall now look at more complicated protocols, starting with one of the most famous namely, the Needham–Schroeder protocol. This protocol was developed in 1978, and is one of most highly studied protocols ever; its fame is due to the fact that even a simple protocol can hide security flaws for a long time. The basic message flows are described as follows, as illustrated in Fig. 2,

$$
\begin{aligned}
A &\longrightarrow S : A, B, N_a, \\
S &\longrightarrow A : \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}, \\
A &\longrightarrow B : \{K_{ab}, A\}_{K_{bs}},
\end{aligned}
$$

$$B \longrightarrow A : \{N_b\}_{K_{ab}},$$
$$A \longrightarrow B : \{N_b - 1\}_{K_{ab}}.$$

We now look at each message in detail, and explain what it does.

- The first message tells S that A wants a key to communicate with B.
- In the second message S generates the session key K_{ab} and sends it back to A. The nonce N_a is included so that A knows this was sent after her request of the first message. The session key is also encrypted under the key K_{bs} for sending to B.
- The third message conveys the session key to B.
- B needs to check that the third message was not a replay. So he needs to know if A is still alive, hence, in the fourth message he encrypts a nonce back to A.
- In the final message, to prove to B that she is still alive, A encrypts a simple function of B's nonce back to B.

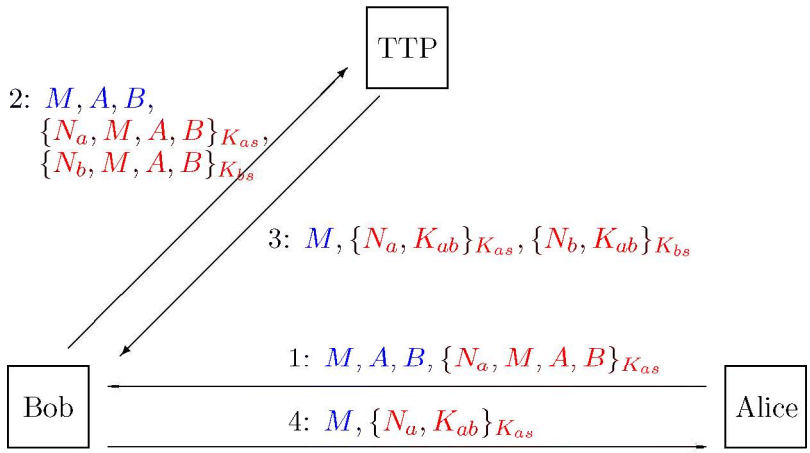
The main problem with the Needham–Schroeder protocol is that B does not know that the key he shares with A is fresh, a fact which was not spotted until some time after the original protocol was published. An adversary who finds an old session transcript can, after finding the old session key by some other means, use the old session transcript in the last three messages involving B. Hence, the adversary can get B to agree to a key with the adversary, which B thinks he is sharing with A.

Note, A and B have their secret session key generated by S and so neither party needs to trust the other to produce 'good' keys. They of course trust S to generate good keys since S is an authority trusted by everyone. In some applications this last assumption is not valid and more involved algorithms, or public key algorithms, are required. In this chapter we shall assume everyone trusts S to perform correctly any action we require of him.

2.4. Otway–Rees Protocol. The Otway–Rees protocol from 1987 is not used that much, but again it is historically important. Like the Needham–Schroeder protocol it does not use synchronized clocks, but again it suffers from a number of problems.

As before two people wish to agree a key using a trusted server S. There are two nonces N_a and N_b used to flag certain encrypted components as recent. In addition a nonce M is used to flag that the current set of communications are linked. The Otway–Rees protocol is shorter than the Needham–Schroeder protocol since it only requires four messages, but the message types are very different. As before the server generates the key K_{ab} for the two parties.

FIGURE 3. Otway–Rees protocol



The message flows in the Otway–Rees protocol are as follows, as illustrated in Fig. 3,

$$
\begin{aligned}
A &\longrightarrow B : M, A, B, \{N_a, M, A, B\}_{K_{as}}, \\
B &\longrightarrow S : M, A, B, \{N_a, M, A, B\}_{K_{as}}, \{N_b, M, A, B\}_{K_{bs}}, \\
S &\longrightarrow B : M, \{N_a, K_{ab}\}_{K_{as}}, \{N_b, K_{ab}\}_{K_{bs}}, \\
B &\longrightarrow A : M, \{N_a, K_{ab}\}_{K_{as}}.
\end{aligned}
$$

Since the protocol does not make use of K_{ab} as an encryption key, neither party knows whether the key is known to each other. We say that Otway–Rees is a protocol which does not offer *key confirmation*. Let us see what the parties do know: A knows that B sent a message containing a nonce N_a which A knows to be fresh, since A originally generated the nonce. So B must have sent a message recently. On the other hand B has been told by the server that A used a nonce, but B has no idea whether this was a replay of an old message.

2.5. Kerberos. We end this section by looking at Kerberos. Kerberos is an authentication system based on symmetric encryption with keys shared with an authentication server; it is based on ideas underlying the Needham–Schroeder protocol. Kerberos was developed at MIT around 1987 as part of Project Athena. A modified version of this original version of Kerberos is now used in Windows 2000.

The network is assumed to consist of clients and a server, where the clients may be users, programs or services. Kerberos keeps a central database of clients including a secret key for each client, hence Kerberos requires a key space of size $O(n)$ if we have n clients. Kerberos is used to provide authentication of one entity to another and to issue session keys to these entities.

In addition Kerberos can run a ticket granting system to enable access control to services and resources. The division between authentication and access is a good idea which we shall see later echoed in SPKI. This division mirrors what happens in real companies. For example, in a company the personnel department administers who you are, whilst the computer department administers what resources you can use. This division is also echoed in Kerberos with an authentication server and a ticket generation server TGS. The TGS gives tickets to enable users to access resources, such as files, printers, etc.

Suppose A wishes to access a resource B. First A logs onto the authentication server using a password. The user A is given a ticket from this server encrypted under her password. This ticket contains a session key K_{as}. She now uses K_{as} to obtain a ticket from the TGS S to access the resource B. The output of the TGS is a key K_{ab} , a timestamp T_S and a lifetime L. The output of the TGS is used to authenticate A in subsequent traffic with B.

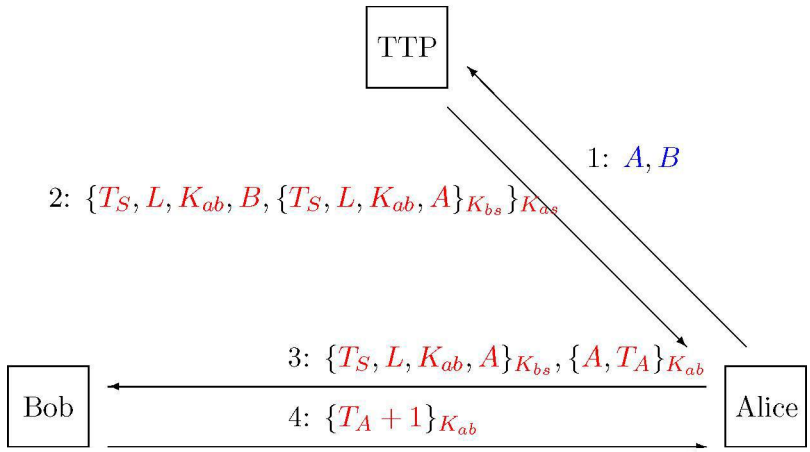
The flows look something like those given in Fig. 4,

$$
\begin{aligned}
A &\longrightarrow S : A, B, \\
S &\longrightarrow A : \{T_S, L, K_{ab}, B, \{T_S, L, K_{ab}, A\}_{K_{bs}}\}_{K_{as}}, \\
A &\longrightarrow B : \{T_S, L, K_{ab}, A\}_{K_{bs}}, \{A, T_A\}_{K_{ab}}, \\
B &\longrightarrow A : \{T_A + 1\}_{K_{ab}}.
\end{aligned}
$$

- The first message is A telling S that she wants to access B.
- If S allows this access then a ticket $\{T_S, L, K_{ab}, A\}$ is created. This is encrypted under K_{bs} and sent to A for forwarding to B. The user A also gets a copy of the key in a form readable by her.
- The user A wants to verify that the ticket is valid and that the resource B is alive. Hence, she sends an encrypted nonce/timestamp T_A to B.
- The resource B sends back the encryption of $T_A + 1$, after checking that the timestamp T_A is recent, thus proving he knows the key and is alive.

We have removed the problems associated with the Needham–Schroeder protocol by using timestamps, but this has created the requirement for synchronized clocks.

FIGURE 4. Kerberos



3. Formal Approaches to Protocol Checking

One can see that the above protocols are very intricate; spotting flaws in them can be a very subtle business. To try and make the design of these protocols more scientific a number of formal approaches have been proposed. The most influential of these is the BAN logic invented by Burrows, Abadi and Needham.

The BAN logic has a large number of drawbacks but was very influential in the design and analysis of symmetric key based key agreement protocols such as Kerberos and the Needham–Schroeder protocol. It has now been supplanted by more complicated logics and formal methods, but it is of historical importance and the study of the BAN logic can still be very instructive for protocol designers.

The main idea of BAN logic is that one should concentrate on what the parties believe is happening. It does not matter what is actually happening, we need to understand exactly what each party can logically deduce, from its own view of the protocol, as to what is actually happening. Even modern approaches to modelling PKI have taken this approach and so we shall now examine the BAN logic in more detail.

We first introduce the notation

- $P| \equiv X$ means P *believes* (or is entitled to believe) X.
  The principal P may act as though X is true.
- $P \triangleleft X$ means P *sees* X.
  Someone has sent a message to P containing X, so P can now read and repeat X.
- $P| \sim X$ means P *once said* X and P believed X when it was said.
  Note this tells us nothing about whether X was said recently or in the distant past.
- $P| \Rightarrow X$ means P has *jurisdiction* over X.
  This means P is an authority on X and should be trusted on this matter.
- $\#X$ means the formula X is *fresh*.
  This is usually used for nonces.
- $P \overset{K}{\leftrightarrow} Q$ means P and Q may use the *shared key* K to communicate.
  The key is assumed good and it will never be discovered by anyone other than P and Q, unless the protocol itself makes this happen.

- $\{X\}_K$, as usual this means X is *encrypted* under the key K.
  The encryption is assumed to be perfect in that X will remain secret unless deliberately disclosed by a party at some other point in the protocol.

In addition, conjunction of statements is denoted by a comma.

There are many postulates, or rules of inference, specified in the BAN logic. We shall only concentrate on the main ones. The format we use to specify rules of inference is as follows:

$$\frac{A, B}{C}$$

which means that if A and B are true then we can conclude C is also true. This is a standard notation used in many areas of logic within computer science.

Message Meaning Rule

$$\frac{A| \equiv A \overset{K}{\leftrightarrow} B, A \triangleleft \{X\}_K}{A| \equiv B| \sim X}.$$

In words, if both

- A believes she shares the key K with B,
- A sees X encrypted under the key K,

we can deduce that A believes that B once said X. Note that this implicitly assumes that A never said X.

Nonce Verification Rule

$$\frac{A| \equiv \#X, A| \equiv B| \sim X}{A| \equiv B| \equiv X}.$$

In words, if both

- A believes X is fresh (i.e. recent),
- A believes B once said X,

then we can deduce that A believes that B still believes X.

Jurisdiction Rule

$$\frac{A| \equiv B| \Rightarrow X, A| \equiv B| \equiv X}{A| \equiv X}.$$

In words, if both

- A believes B has jurisdiction over X, i.e. A trusts B on X,
- A believes B believes X,

then we conclude that A also believes X.

Other Rules

The belief operator and conjunction can be manipulated as follows:

$$\frac{P| \equiv X, P| \equiv Y}{P| \equiv (X, Y)}, \quad \frac{P| \equiv (X, Y)}{P| \equiv X}, \quad \frac{P| \equiv Q| \equiv (X, Y)}{P| \equiv Q| \equiv X}.$$

A similar rule also applies to the 'once said' operator

$$\frac{P| \equiv Q| \sim (X, Y)}{P| \equiv Q| \sim X}.$$

Note that $P| \equiv Q| \sim X$ and $P| \equiv Q| \sim Y$ does not imply $P| \equiv Q| \sim (X, Y)$, since that would imply X and Y were said at the same time. Finally, if part of a formula is fresh then so is the whole formula

$$\frac{P| \equiv \#X}{P| \equiv \#(X, Y)}.$$

We wish to analyse a key agreement protocol between A and B using the BAN logic. But what is the goal of such a protocol? The minimum we want to achieve is

$$A| \equiv A \overset{K}{\leftrightarrow} B \text{ and } B| \equiv A \overset{K}{\leftrightarrow} B,$$

i.e. both parties believe they share a secret key with each other.

However, we could expect to achieve more, for example

$$A| \equiv B| \equiv A \overset{K}{\leftrightarrow} B \text{ and } B| \equiv A| \equiv A \overset{K}{\leftrightarrow} B,$$

which is called key confirmation. In words, we may want to achieve that, after the protocol has run, A is assured that B knows he is sharing a key with A, and it is the same key A believes she is sharing with B.

Before analysing a protocol using the BAN logic we convert the protocol into logical statements. This process is called *idealization*, and is the most error prone part of the procedure since it cannot be automated. We also need to specify the assumptions, or axioms, which hold at the beginning of the protocol.

To see this in 'real life' we analyse the Wide-Mouth Frog protocol for key agreement using synchronized clocks.

3.1. Wide-Mouth Frog Protocol. Recall the Wide-Mouth Frog protocol

$$A \longrightarrow S : A, \{T_a, B, K_{ab}\}_{K_{as}},$$
$$S \longrightarrow B : \{T_s, A, K_{ab}\}_{K_{bs}}.$$

This becomes the idealized protocol

$$A \longrightarrow S : \{T_a, A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{as}},$$
$$S \longrightarrow B : \{T_s, A| \equiv A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}.$$

One should read the idealization of the first message as telling S that

- T_a is a timestamp/nonce,
- K_{ab} is a key which is meant as a key to communicate with B.

So what assumptions exist at the start of the protocol? Clearly A, B and S share secret keys which in BAN logic becomes

$$A| \equiv A \overset{K_{as}}{\leftrightarrow} S, \qquad S| \equiv A \overset{K_{as}}{\leftrightarrow} S,$$
$$B| \equiv B \overset{K_{bs}}{\leftrightarrow} S, \qquad S| \equiv B \overset{K_{bs}}{\leftrightarrow} S.$$

There are a couple of nonce assumptions,

$$S| \equiv \#T_a \text{ and } B| \equiv \#T_s.$$

Finally, we have the following three assumptions

- B trusts A to invent good keys,

$$B| \equiv (A| \Rightarrow A \overset{K_{ab}}{\leftrightarrow} B),$$

- B trusts S to relay the key from A,

$$B| \equiv (S| \Rightarrow A| \equiv A \overset{K_{ab}}{\leftrightarrow} B),$$

- A knows the session key in advance,

$$A| \equiv A \overset{K_{ab}}{\leftrightarrow} B.$$

Notice how these last three assumptions specify the problems we associated with this protocol in the earlier section.

Using these assumptions we can now analyse the protocol. Let us see what we can deduce from the first message

$$A \longrightarrow S : \{T_a, A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{as}}.$$

- Since S sees the message encrypted under K_{as} he can deduce that A said the message.
- Since T_a is believed by S to be fresh he concludes the whole message is fresh.
- Since the whole message is fresh, S concludes that A currently believes the whole of it.
- S then concludes

$$S| \equiv A| \equiv A \overset{K_{ab}}{\leftrightarrow} B,$$

which is what we need to conclude so that S can send the second message of the protocol.

We now look at what happens when we analyse the second message

$$S \longrightarrow B : \{T_s, A| \equiv A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}.$$

- Since B sees the message encrypted under K_{bs} he can deduce that S said the message.
- Since T_s is believed by B to be fresh he concludes the whole message is fresh.
- Since the whole message is fresh, B concludes that S currently believes the whole of it.
- So B believes that S believes the second part of the message.
- But B believes S has authority on whether A knows the key and B believes A has authority to generate the key.

So we conclude

$$B| \equiv A \overset{K_{ab}}{\leftrightarrow} B$$

and

$$B| \equiv A| \equiv A \overset{K_{ab}}{\leftrightarrow} B.$$

Combining with our axiom $A| \equiv A \overset{K_{ab}}{\leftrightarrow} B$ we conclude that the key agreement protocol is sound. The only requirement we have not met is that

$$A| \equiv B| \equiv A \overset{K_{ab}}{\leftrightarrow} B,$$

i.e. A does not achieve confirmation that B has received the key.

Notice what the application of the BAN logic has done is to make the axioms clearer, so it is easier to compare which assumptions each protocol needs to make it work. In addition it clarifies what the result of running the protocol is from all parties' points of view.

Chapter Summary

- Distributing secret keys used for symmetric ciphers can be a major problem.
- A number of key agreement protocols exist based on a trusted third party and symmetric encryption algorithms. These protocols require long-term keys to have been already established with the TTP, they may also require some form of clock synchronization.
- Various logics exist to analyse such protocols. The most influential of these has been the BAN logic. These logics help to identify explicit assumptions and problems associated with each protocol.

Further Reading

The paper by Burrows, Abadi and Needham is a very readable introduction to the BAN logic and a number of key agreement protocols, much of our treatment is based on this paper. For another, more modern, approach to protocol checking see the book by Ryan et. al., this covers an approach based on the CSP process algebra.

M. Burrows, M. Abadi and R. Needham. *A Logic of Authentication.* Digital Equipment Corporation, SRC Research Report 39, 1990.

P. Ryan, S. Schneider, M. Goldsmith, G. Lowe and B. Ruscoe. *Modelling and analysis of security protocols.* Addison–Wesley, 2001.

CHAPTER 10

Hash Functions and Message Authentication Codes

Chapter Goals

- To understand the properties of cryptographic hash functions.
- To understand how existing deployed hash functions work.
- To examine the workings of message authentication codes.

1. Introduction

In many situations we do not wish to protect the confidentiality of information, we simply wish to ensure the integrity of information. That is we want to gaurantee that data has not been tampered with. In this chapter we look at two mechanisms for this, the first using cryptographic hash functions is for when we want to gaurantee integrity of information after the application of the function. A cryptographic hash function is usually used as a component of another scheme, since the integrity is not bound to any entity. The other mechanism we look at is the use of a message authentication code. These act like a keyed version of a hash function, they are a symmetric key technique which enables the holders of a symmetric key to agree that only they could have produced the authentication code on a given message.

Hash functions can also be considered as a special type of manipulation detection code, or MDC. For example a hash function can be used to protect the integrity of a large file, as used in some virus protection products. The hash value of the file contents is computed and then either stored in a secure place (e.g. on a floppy in a safe) or the hash value is put in a file of similar values which is then digitally signed to stop future tampering.

Both hash functions and MACs will be used extensively later in other schemes. In particular cryptographic hash functions will be used to compress large messages down to smaller ones to enable efficient digital signature algorithms. Another use of hash functions is to produce, in a deterministic manner, random data from given values. We shall see this application when we build elaborate and "provably secure" encryption schemes later on in the book.

2. Hash Functions

A cryptographic hash function h is a function which takes arbitrary length bit strings as input and produces a fixed length bit string as output, the output is often called a hashcode or hash value. Hash functions are used a lot in computer science, but the crucial difference between a standard hash function and a cryptographic hash function is that a cryptographic hash function should at least have the property of being one-way. In other words given any string y from the range of h, it should be computationally infeasible to find any value x in the domain of h such that

$$h(x) = y.$$

Another way to describe a hash function which has the one-way property is that it is preimage resistant. Given a hash function which produces outputs of n bits, we would like a function for which finding preimages requires $O(2^n)$ time.

In practice we need something more than the one-way property. A hash function is called collision resistant if it is infeasible to find two distinct values x and x' such that

$$h(x) = h(x').$$

It is harder to construct collision resistant hash functions than one-way hash functions due to the birthday paradox. To find a collision of a hash function f, we can keep computing

$$f(x_1), f(x_2), f(x_3), \ldots$$

until we get a collision. If the function has an output size of n bits then we expect to find a collision after $O(2^{n/2})$ iterations. This should be compared with the number of steps needed to find a preimage, which should be $O(2^n)$ for a well-designed hash function. Hence to achieve a security level of 80 bits for a collision resistant hash function we need roughly 160 bits of output.

But still that is not enough; a cryptographic hash function should also be second preimage resistant. This is the property that given m it should be hard to find an $m' \neq m$ with $h(m') = h(m)$. Whilst this may look like collision resistance, it is actually related more to preimage resistance. In particular a cryptographic hash function with n-bit outputs should require $O(2^n)$ operations before one can find a second preimage.

In summary a cryptographic hash function needs to satisfy the following three properties:

(1) **Preimage Resistant:** It should be hard to find a message with a given hash value.
(2) **Collision Resistant:** It should be hard to find two messages with the same hash value.
(3) **Second Preimage Resistant:** Given one message it should be hard to find another message with the same hash value.

But how are these properties related. We can relate these properties using reductions.

LEMMA 10.1. *Assuming a function is preimage resistant* for every element of the range of h *is a weaker assumption than assuming it either collision resistant or second preimage resistant.*

PROOF. Suppose h is a function and let $\mathcal{O}$ denote an oracle which on input of y finds an x such that $h(x) = y$, i.e. $\mathcal{O}$ is an oracle which breaks the preimage resistance of the function h.

Using $\mathcal{O}$ we can then find a collision in h by pulling x at random and then computing $y = h(x)$. Passing y to the oracle $\mathcal{O}$ will produce a value x' such that $y = h(x')$. Since h is assumed to have infinite domain, it is unlikely that we have $x = x'$. Hence, we have found a collision in h.

A similar argument applies to breaking the second preimage resistance of h. □

However, one can construct hash functions which are collision resistant but are not one-way for some of the range of h. As an example, let $g(x)$ denote a collision resistant hash function with outputs of bit length n. Now define a new hash function $h(x)$ with output size $n+1$ bits as follows:

$$h(x) = \begin{cases} 0\|x & \text{If } |x| = n, \\ 1\|g(x) & \text{Otherwise.} \end{cases}$$

The function $h(x)$ is clearly collision resistant, as we have assumed $g(x)$ is collision resistant. But the function $h(x)$ is not preimage resistant as one can invert it on any value in the range which starts with a zero bit. So even though we can invert the function $h(x)$ on some of its input we are unable to find collisions.

LEMMA 10.2. *Assuming a function is second preimage resistant is a weaker assumption than assuming it is collision resistant.*

PROOF. Assume we are given an oracle $\mathcal{O}$ which on input of x will find x' such that $x \neq x'$ and $h(x) = h(x')$. We can clearly use $\mathcal{O}$ to find a collision in h by choosing x at random. □

3. Designing Hash Functions

To be effectively collision free a hash value should be at least 128 bits long, for applications with low security, but preferably its output should be 160 bits long. However, the input size should be bit strings of (virtually) infinite length. In practice designing functions of infinite domain is hard, hence usually one builds a so called compression function which maps bits strings of length s into bit strings of length n, for $s > n$, and then chains this in some way so as to produce a function on an infinite domain. We have seen such a situation before when we considered modes of operation of block ciphers.

We first discuss the most famous chaining method, namely the Merkle–Damgård construction, and then we go on to discuss designs for the compression function.

3.1. Merkle–Damgård Construction. Suppose f is a compression function from s bits to n bits, with $s > n$, which is believed to be collision resistant. We wish to use f to construct a hash function h which takes arbitrary length inputs, and which produces hash codes of n bits in length. The resulting hash function should be collision resistant. The standard way of doing this is to use the Merkle–Damgård construction described in Algorithm 10.1.

Algorithm 10.1: Merkle–Damgård Construction

$l = s - n$
Pad the input message m with zeros so that it is a multiple of l bits in length
Divide the input m into t blocks of l bits long, $m_1, \ldots, m_t$
Set H to be some fixed bit string of length n.
for $i = 1$ **to** t **do**
    $H = f(H \| m_i)$
end
return (H)

In this algorithm the variable H is usually called the *internal state* of the hash function. At each iteration this internal state is updated, by taking the current state and the next message block and applying the compression function. At the end the internal state is output as the result of the hash function.

Algorithm 10.1 describes the basic Merkle–Damgård construction, however it is almost always used with so called *length strengthening*. In this variant the input message is preprocessed by first padding with zero bits to obtain a message which has length a multiple of l bits. Then a final block of l bits is added which encodes the original length of the unpadded message in bits. This means that the construction is limited to hashing messages with length less than 2^l bits.

To see why the strengthening is needed consider a "baby" compression function f which maps bit strings of length 8 into bit strings of length 4 and then apply it to the two messages

$$m_1 = 0b0, \ m_2 = 0b00.$$

Whilst the first message is one bit long and the second message is two bits long, the output of the basic Merkle–Damgård construction will be

$$h(m_1) = f(0b00000000) = h(m_2),$$

i.e. we obtain a collision. However, with the strengthened version we obtain the following hash values in our baby example

$$\begin{aligned} h(m_1) &= f(f(0b00000000) \| 0b0001), \\ h(m_2) &= f(f(0b00000000) \| 0b0010). \end{aligned}$$

These last two values will be different unless we just happen to have found a collision in f.

Another form of length strengthening is to add a single one bit onto the data to signal the end of a message, pad with zeros, and then apply the hash function. Our baby example in this case would become

$$\begin{aligned} h(m_1) &= f(0b01000000), \\ h(m_2) &= f(0b00100000). \end{aligned}$$

Yet another form is to combine this with the previous form of length strengthening, so as to obtain

$$\begin{aligned} h(m_1) &= f(f(0b01000000)\|0b0001), \\ h(m_2) &= f(f(0b00100000)\|0b0010). \end{aligned}$$

3.2. The MD4 Family. A basic design principle when designing a compression function is that its output should produce an avalanche effect, in other words a small change in the input produces a large and unpredictable change in the output. This is needed so that a signature on a cheque for 30 pounds cannot be altered into a signature on a cheque for 30 000 pounds, or vice versa. This design principle is typified in the MD4 family which we shall now describe.

Several hash functions are widely used, they are all iterative in nature. The three most widely deployed are MD5, RIPEMD-160 and SHA-1. The MD5 algorithm produces outputs of 128 bits in size, whilst RIPEMD-160 and SHA-1 both produce outputs of 160 bits in length. Recently NIST has proposed a new set of hash functions called SHA-256, SHA-384 and SHA-512 having outputs of 256, 384 and 512 bits respectively, collectively these algorithms are called SHA-2. All of these hash functions are derived from an earlier simpler algorithm called MD4.

The seven main algorithms in the MD4 family are

- **MD4**: This has 3 rounds of 16 steps and an output bitlength of 128 bits.
- **MD5**: This has 4 rounds of 16 steps and an output bitlength of 128 bits.
- **SHA-1**: This has 4 rounds of 20 steps and an output bitlength of 160 bits.
- **RIPEMD-160**: This has 5 rounds of 16 steps and an output bitlength of 160 bits.
- **SHA-256**: This has 64 rounds of single steps and an output bitlength of 256 bits.
- **SHA-384**: This is identical to SHA-512 except the output is truncated to 384 bits.
- **SHA-512**: This has 80 rounds of single steps and an output bitlength of 512 bits.

We discuss MD4 and SHA-1 in detail, the others are just more complicated versions of MD4, which we leave to the interested reader to look up in the literature.

In recent years a number of weaknesses have been found in almost all of the early hash functions in the MD4 family, for example MD4, MD5 and SHA-1. Hence, it is wise to move all application to use the SHA-2 algorithms.

3.3. MD4. In MD4 there are three bit-wise functions of three 32-bit variables

$$\begin{aligned} f(u,v,w) &= (u \wedge v) \vee ((\neg u) \wedge w), \\ g(u,v,w) &= (u \wedge v) \vee (u \wedge w) \vee (v \wedge w), \\ h(u,v,w) &= u \oplus v \oplus w. \end{aligned}$$

Throughout the algorithm we maintain a current hash state

$$(H_1, H_2, H_3, H_4)$$

of four 32-bit values initialized with a fixed initial value,

$$\begin{aligned} H_1 &= \texttt{0x67452301}, \\ H_2 &= \texttt{0xEFCDAB89}, \\ H_3 &= \texttt{0x98BADCFE}, \\ H_4 &= \texttt{0x10325476}. \end{aligned}$$

There are various fixed constants (y_i, z_i, s_i), which depend on each round. We have

$$y_j = \begin{cases} 0 & 0 \leq j \leq 15, \\ \texttt{0x5A827999} & 16 \leq j \leq 31, \\ \texttt{0x6ED9EBA1} & 32 \leq j \leq 47, \end{cases}$$

and the values of z_i and s_i are given by following arrays,

$$\begin{aligned} z_{0\ldots15} &= [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15], \\ z_{16\ldots31} &= [0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15], \\ z_{32\ldots47} &= [0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15], \\ s_{0\ldots15} &= [3,7,11,19,3,7,11,19,3,7,11,19,3,7,11,19], \\ s_{16\ldots31} &= [3,5,9,13,3,5,9,13,3,5,9,13,3,5,9,13], \\ s_{32\ldots47} &= [3,9,11,15,3,9,11,15,3,9,11,15,3,9,11,15]. \end{aligned}$$

The data stream is loaded 16 words at a time into X_j for $0 \leq j < 16$. The length strengthening method used is to first append a one bit to the message, to signal its end and then to pad with zeros to a multiple of the block length. Finally the number of bits of the message is added as a seperate final block.

We then execute the steps in Algorithm 10.2 for each 16 words entered from the data stream.

Algorithm 10.2: MD4 Overview

$(A, B, C, D) = (H_1, H_2, H_3, H_4)$
Execute Round 1
Execute Round 2
Execute Round 3
$(H_1, H_2, H_3, H_4) = (H_1 + A, H_2 + B, H_3 + C, H_4 + D)$

After all data has been read in, the output is the concatenation of the final value of

$$H_1, H_2, H_3, H_4.$$

The details of the rounds are given by Algorithm 10.3 where $\lll$ denotes a bit-wise rotate to the left:

Algorithm 10.3: Description of the MD4 round functions

Round 1
for $j = 0$ **to** 15 **do**
  $t = A + f(B, C, D) + X_{z_j} + y_j$
  $(A, B, C, D) = (D, t \lll s_j, B, C)$
end
Round 2
for $j = 16$ **to** 31 **do**
  $t = A + g(B, C, D) + X_{z_j} + y_j$
  $(A, B, C, D) = (D, t \lll s_j, B, C)$
end
Round 3
for $j = 32$ **to** 47 **do**
  $t = A + h(B, C, D) + X_{z_j} + y_j$
  $(A, B, C, D) = (D, t \lll s_j, B, C)$
end

3.4. SHA-1. We use the same bit-wise functions f, g and h as in MD4. For SHA-1 the internal state of the algorithm is a set of five, rather than four, 32-bit values

$$(H_1, H_2, H_3, H_4, H_5).$$

These are assigned with the initial values

$$\begin{aligned} H_1 &= \texttt{0x67452301}, \\ H_2 &= \texttt{0xEFCDAB89}, \\ H_3 &= \texttt{0x98BADCFE}, \\ H_4 &= \texttt{0x10325476}, \\ H_5 &= \texttt{0xC3D2E1F0}, \end{aligned}$$

We now only define four round constants y_1, y_2, y_3, y_4 via

$$\begin{aligned} y1 &= \texttt{0x5A827999}, \\ y2 &= \texttt{0x6ED9EBA1}, \\ y3 &= \texttt{0x8F1BBCDC}, \\ y4 &= \texttt{0xCA62C1D6}. \end{aligned}$$

The data stream is loaded 16 words at a time into X_j for $0 \leq j < 16$, although note the internals of the algorithm uses an expanded version of X_j with indices from 0 to 79. The length strengthening method used is to first append a one bit to the message, to signal its end and then to pad with zeros to a multiple of the block length. Finally the number of bits of the message is added as a seperate final block.

We then execute the steps in Algorithm 10.4 for each 16 words entered from the data stream. The details of the rounds are given by Algorithm 10.5.

Algorithm 10.4: SHA-1 Overview

$(A, B, C, D, E) = (H_1, H_2, H_3, H_4, H_5)$
/* Expansion */
for $j = 16$ **to** 79 **do**
    $X_j = ((X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}) \lll 1)$
end
Execute Round 1
Execute Round 2
Execute Round 3
Execute Round 4
$(H_1, H_2, H_3, H_4, H_5) = (H_1 + A, H_2 + B, H_3 + C, H_4 + D, H_5 + E)$

Note the one bit left rotation in the expansion step, an earlier algorithm called SHA(now called SHA-0) was initially proposed by NIST which did not include this one bit rotation. This was however soon replaced by the new algorithm SHA-1. It turns out that this single one bit rotation improves the security of the resulting hash function quite a lot.

After all data has been read in, the output is the concatenation of the final value of

$$H_1, H_2, H_3, H_4, H_5.$$

3.5. Hash Functions and Block Ciphers. One can also make a hash function out of an n-bit block cipher, E_K. There are a number of ways of doing this, all of which make use of a constant public initial value IV. Some of the schemes also make use of a function g which maps n-bit inputs to keys.

Algorithm 10.5: Description of the SHA-1 round functions

Round 1
for $j = 0$ **to** 19 **do**
  $t = (A \lll 5) + f(B, C, D) + E + X_j + y_1$
  $(A, B, C, D, E) = (t, A, B \lll 30, C, D)$
end
Round 2
for $j = 20$ **to** 39 **do**
  $t = (A \lll 5) + h(B, C, D) + E + X_j + y_2$
  $(A, B, C, D, E) = (t, A, B \lll 30, C, D)$
end
Round 3
for $j = 40$ **to** 59 **do**
  $t = (A \lll 5) + g(B, C, D) + E + X_j + y_3$
  $(A, B, C, D, E) = (t, A, B \lll 30, C, D)$
end
Round 4
for $j = 60$ **to** 79 **do**
  $t = (A \lll 5) + h(B, C, D) + E + X_j + y_4$
  $(A, B, C, D, E) = (t, A, B \lll 30, C, D)$
end

We first pad the message to be hashed and divide it into blocks

$$x_0, x_1, \ldots, x_t,$$

of size either the block size or key size of the underlying block cipher, the exact choice of size depending on the exact definition of the hash function being created. The output hash value is then the final value of H_i in the following iteration

$$H_0 = IV,$$
$$H_i = f(x_i, H_{i-1}).$$

The exact definition of the function f depends on the scheme being used. We present just three, although others are possible.

- **Matyas–Meyer–Oseas hash**

$$f(x_i, H_{i-1}) = E_{g(H_{i-1})}(x_i) \oplus x_i.$$

- **Davies–Meyer hash**

$$f(x_i, H_{i-1}) = E_{x_i}(H_{i-1}) \oplus H_{i-1}.$$

- **Miyaguchi–Preneel hash**

$$f(x_i, H_{i-1}) = E_{g(H_{i-1})}(x_i) \oplus x_i \oplus H_{i-1}.$$

4. Message Authentication Codes

Given a message and its hash code, as output by a cryptographic hash function, ensures that data has not been tampered with between the execution of the hash function and its verification, by recomputing the hash. However, using a hash function in this way requires the hash code itself to be protected in some way, by for example a digital signature, as otherwise the hash code itself could be tampered with.

To avoid this problem one can use a form of keyed hash function called a message authentication code, or MAC. This is a symmetric key algorithm in that the person creating the code and the person verifying it both require the knowledge of a shared secret.

Suppose two parties, who share a secret key, wish to ensure that data transmitted between them has not been tampered with. They can then use the shared secret key and a keyed algorithm to produce a check-value, or MAC, which is sent with the data. In symbols we compute

$$\text{code} = MAC_k(m)$$

where

- MAC is the check function,
- k is the secret key,
- m is the message.

Note we do not assume that the message is secret, we are trying to protect data integrity and not confidentiality. If we wish our message to remain confidential then we should encrypt it before applying the MAC. After performing the encryption and computing the MAC, the user transmits

$$e_{k_1}(m) \| MAC_{k_2}(e_{k_1}(m)) .$$

This is a form of encryption called a data encapsulation mechanism, or DEM for short. Note, that different keys are used for the encryption and the MAC part of the message and that the MAC is applied to the ciphertext and not the message.

Before we proceed on how to construct MAC functions it is worth pausing to think about what security properties we require. We would like that only people who know the shared secret are able to both produce new MACs or verify existing MACs. In particular it should be hard given a MAC on a message to produce a MAC on a new message.

4.1. Producing MACs from hash functions. A collision-free cryptographic hash function can also be used as the basis of a MAC. The first idea one comes up with to construct such a MAC is to concaternate the key with the message and then apply the hash function. For example

$$MAC_k(M) = h(k \| M).$$

However, this is not a good idea since almost all hash functions are created using methods like the Merkle–Damgård construction. This allows us to attack such a MAC as follows: We assume that first that the non-length strengthed Merkle–Damgård construction is used with compression function f. Suppose one obtains the MAC c_1 on the t block message m_1

$$c_1 = MAC_k(m_1) = h(k \| m_1)$$

We can then, without knowledge of k compute the MAC c_2 on the $t+1$ block message $m_1 \| m_2$ for any m_2 of one block in length, via

$$\begin{aligned} c_2 &= MAC_k(m_1 \| m_2) \\ &= f(c_1 \| m_2). \end{aligned}$$

Clearly this attack can be extended to a appending an m_2 of arbitrary length. Hence, we can also apply it to the length strengthed version. If we let m_1 denote a t block message and let b denote the block which encodes the bit length of m_1 and we let m_2 denote an arbitrary new block, then from the MAC of the message m_1 one can obtain the MAC of the message

$$m_1 \| b \| m_2.$$

Having worked out that prepending a key to a message does not give a secure MAC, one might be led to try appending the key after the message as in

$$MAC_k(M) = h(M \| k).$$

Again we now can make use of the Merkle–Damgård construction to produce an attack. We first, without knowledge of k, find via a birthday attack on the hash function h two equal length messages m_1 and m_2 which hash to the same values:

$$h(m_1) = h(m_2).$$

We now try to obtain the legitimate MAC c_1 on the message m_1. From, this we can deduce the MAC on the message m_2 via

$$\begin{aligned} MAC_k(m_2) &= h(m_2\|k) \\ &= f\left(h(m_2)\|k\right) \\ &= f\left(h(m_1)\|k\right) \\ &= h(m_1\|k) \\ &= MAC_k(m_1) \\ &= c_1. \end{aligned}$$

assuming k is a single block in length and the non-length strengthened version is used. Both of these assumptions can be relaxed, the details of which we leave to the reader.

To produce a secure MAC from a hash function one needs to be a little more clever. A MAC, called HMAC, occurring in a number of standards documents works as follows:

$$HMAC = h(k\|p_1\|h(k\|p_2\|M)),$$

where p_1 and p_2 are strings used to pad out the input to the hash function to a full block.

4.2. Producing MACs from block ciphers. Apart from ensuring the confidentiality of messages, block ciphers can also be used to protect the integrity of data. There are various types of MAC schemes based on block ciphers, but the best known and most widely used by far are the CBC-MACs. These are generated by a block cipher in CBC Mode. CBC-MACs are the subject of various international standards dating back to the early 1980s. These early standards specify the use of DES in CBC mode to produce a MAC, although one could really use any block cipher in place of DES.

Using an n-bit block cipher to give an m-bit MAC, where $m \leq n$, is done as follows:

- The data is padded to form a series of n-bit blocks.
- The blocks are encrypted using the block cipher in CBC Mode.
- Take the final block as the MAC, after an optional postprocessing stage and truncation (if $m < n$).

Hence, if the n-bit data blocks are

$$m_1, m_2, \ldots, m_q$$

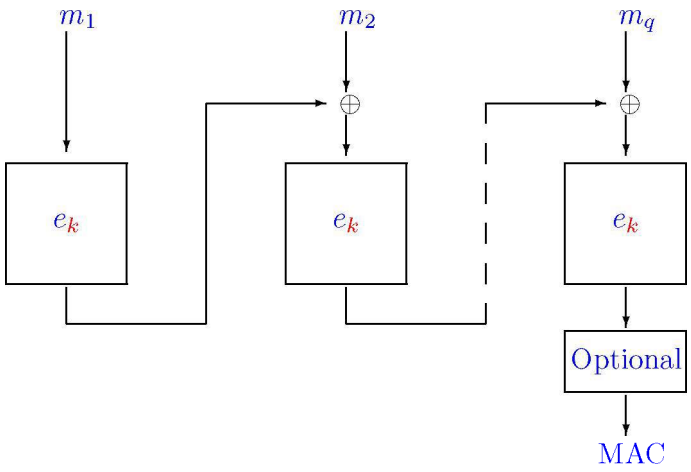
then the MAC is computed by first setting $I_1 = m_1$ and $O_1 = e_k(I_1)$ and then performing the following for $i = 2, 3, \ldots, q$

$$\begin{aligned} I_i &= m_i \oplus O_{i-1}, \\ O_i &= e_k(I_i). \end{aligned}$$

The final value O_q is then subject to an optional processing stage. The result is then truncated to m bits to give the final MAC. This is all summarized in Fig. 1.

Just as with hash functions one needs to worry about how one pads the message before applying the CBC-MAC. The three main padding methods proposed in the standards, are as follows, and are equivalent to those already considered for hash functions:

FIGURE 1. CBC-MAC: Flow diagram



- **Method 1:** Add as many zeros as necessary to make a whole number of blocks. This method has a number of problems associated to it as it does not allow the detection of the addition or deletion of trailing zeros, unless the message length is known.
- **Method 2:** Add a single one to the message followed by as many zeros as necessary to make a whole number of blocks. The addition of the extra bit is used to signal the end of the message, in case the message ends with a string of zeros.
- **Method 3:** As method one but also add an extra block containing the length of the unpadded message.

Before we look at the "optional" post-processing steps let us first see what happens if no post-processing occurs. We first look at an attack which uses padding method one. Suppose we have a MAC M on a message

$$m_1, m_2, \ldots, m_q,$$

consisting of a whole number of blocks. Then one can the MAC M is also the MAC of the double length message

$$m_1, m_2, \ldots, m_q, M \oplus m_1, m_2, m_3, \ldots, m_q.$$

To see this notice that the input to the $(q+1)$'st block cipher envocation is equal to the value of the MAC on the original message, namely M, xor'd with the $(q+1)$'st block of the new message, namely $M \oplus m_1$. Thus the input to the $(q+1)$'st cipher envocation is equalk to m_1, and so the MAC on the double length message is also equal to M.

One could suspect that if you used padding method three above then attacks would be impossible. Let b denote the block length of the cipher and let $\mathbb{P}(n)$ denote the encoding within a block of the number n. To MAC a single block message m_1 one then computes

$$M_1 = e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right).$$

Suppose one obtains the MAC's M_1 and M_2 on the single block messages m_1 and m_2. Then one requests the MAC on the three block message

$$m_1, \mathbb{P}(b), m_3$$

for some new block m_3. Suppose the recieved MAC is then equal to M_3, i.e.

$$M_3 = e_k\left(e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right) \oplus m_3\right) \oplus \mathbb{P}(3b)\right).$$

Now also consider the MAC on the three block message

$$m_2, \mathbb{P}(b), m_3 \oplus M_1 \oplus M_2.$$

This MAC is equal to M_3', where

$$\begin{aligned} M_3' &= e_k\left(e_k\left(e_k\left(e_k(m_2)\oplus\mathbb{P}(b)\right)\oplus m_3\oplus M_1\oplus M_2\right)\oplus\mathbb{P}(3b)\right) \\ &= e_k\left(e_k\left(e_k\left(e_k(m_2)\oplus\mathbb{P}(b)\right)\oplus m_3\oplus e_k\left(e_k(m_1)\oplus\mathbb{P}(b)\right)\oplus e_k\left(e_k(m_2)\oplus\mathbb{P}(b)\right)\right)\right. \\ &\quad \left.\oplus\mathbb{P}(3b)\right) \\ &= e_k\left(e_k\left(m_3\oplus e_k\left(e_k(m_1)\oplus\mathbb{P}(b)\right)\right)\oplus\mathbb{P}(3b)\right) \\ &= e_k\left(e_k\left(e_k\left(e_k(m_1)\oplus\mathbb{P}(b)\right)\oplus m_3\right)\oplus\mathbb{P}(3b)\right) \\ &= M_3. \end{aligned}$$

Hence, we see that on their own the non-trivial padding methods do not protect against MAC forgery attacks. This is one of the reasons for introducing the post processing steps. There are two popular post-processing steps, designed to make it more difficult for the cryptanalyst to perform an exhaustive key search and to protect against attacks such as the ones explained above:

(1) Choose a key k_1 and compute

$$O_q = e_k\left(d_{k_1}(O_q)\right).$$

(2) Choose a key k_1 and compute

$$O_q = e_{k_1}(O_q).$$

Both of these post-processing steps were invented when DES was the dominant cipher, and in such a situation the first of these is equivalent to processing the final block of the message using the 3DES algorithm.

Chapter Summary

- Hash functions are required which are both preimage, collision and second-preimage resistant.
- Due to the birthday paradox the output of the hash function should be at least twice the size of what one believes to be the limit of the computational ability of the attacker.
- More hash functions are iterative in nature, although most of the currently deployed ones have recently shown to be weaker than expected.
- A message authentication code is in some sense a keyed hash function.
- MACs can be created out of either block ciphers or hash functions.

Further Reading

A detailed description of both SHA-1 and the SHA-2 algorithms can be found in the FIPS standard below, this includes a set of test vectors as well. The recent work on the analysis of SHA-1, and references to the earlier attacks on MD4 and MD5 can be found in the papers og Wang et. al., of which we list only one below.

FIPS PUB 180-2, *Secure Hash Standard (including SHA-1, SHA-256, SHA-384, and SHA-512)*. NIST, 2005.

X. Wang, Y.L. Yin and H. Yu. *Finding Collisions in the Full SHA-1* In Advances in Cryptology – CRYPTO 2005, Springer-Verlag LNCS 3621, pp 17-36, 2005.

Part 3

Public Key Encryption and Signatures

Public key techniques were originally invented to solve the key distribution problem and to provide authenticity. They have many advantages over symmetric systems, the main one is that they do not require two communicating parties to know each other before encrypted communication can take place. In addition the use of digital signatures allows users to sign digital data such as electronic orders or money transfers. Hence, public key technology is one of the key enabling technologies for e-commerce and a digital society.

CHAPTER 11

Basic Public Key Encryption Algorithms

Chapter Goals

- To learn about public key encryption and the hard problems on which it is based.
- To understand the RSA algorithm and the assumptions on which its security relies.
- To understand the ElGamal encryption algorithm and it assumptions.
- To learn about the Rabin encryption algorithm and its assumptions.

1. Public Key Cryptography

Recall that in symmetric key cryptography each communicating party needed to have a copy of the same secret key. This led to a very difficult key management problem. In public key cryptography we replace the use of identical keys with two keys, one public and one private.

The public key can be published in a directory along with the user's name. Anyone who then wishes to send a message to the holder of the associated private key will take the public key, encrypt a message under it and send it to the owner of the corresponding private key. The idea is that only the holder of the private key will be able to decrypt the message. More clearly, we have the transforms

$$\text{Message} + \text{Alice's public key} = \text{Ciphertext},$$
$$\text{Ciphertext} + \text{Alice's private key} = \text{Message}.$$

Hence anyone with Alice's public key can send Alice a secret message. But only Alice can decrypt the message, since only Alice has the corresponding private key.

Public key systems work because the two keys are linked in a mathematical way, such that knowing the public key tells you nothing about the private key. But knowing the private key allows you to unlock information encrypted with the public key. This may seem strange, and will require some thought and patience to understand. The concept was so strange it was not until 1976 that anyone thought of it. The idea was first presented in the seminal paper of Diffie and Hellman entitled *New Directions in Cryptography*. Although Diffie and Hellman invented the concept of public key cryptography it was not until a year or so later that the first (and most successful) system, namely RSA, was invented.

The previous paragraph is how the 'official' history of public key cryptography goes. However, in the late 1990s an unofficial history came to light. It turned out that in 1969, over five years before Diffie and Hellman invented public key cryptography, a cryptographer called James Ellis, working for the British government's communication headquarters GCHQ, invented the concept of public key cryptography (or non-secret encryption as he called it) as a means of solving the key distribution problem. Ellis, just like Diffie and Hellman, did not however have a system.

The problem of finding such a public key encryption system was given to a new recruit to GCHQ called Clifford Cocks in 1973. Within a day Cocks had invented what was essentially the RSA algorithm, although a full four years before Rivest, Shamir and Adleman. In 1974 another employee at GCHQ, Malcolm Williamson, invented the concept of Diffie–Hellman key exchange,

which we shall return to in Chapter 14. Hence, by 1974 the British security services had already discovered the main techniques in public key cryptography.

There is a surprisingly small number of ideas behind public key encryption algorithms, which may explain why once Diffie and Hellman or Ellis had the concept of public key encryption, an invention of essentially the same cipher, i.e. RSA, came so quickly. There are so few ideas because we require a mathematical operation which is easy to do one way, i.e. encryption, but which is hard to do the other way, i.e. decryption, without some special secret information, namely the private key. Such a mathematical function is called a trapdoor one-way function, since it is effectively a one-way function unless one knows the key to the trapdoor.

Luckily there are a number of possible one-way functions which have been well studied, such as factoring integers, computing discrete logarithms or computing square roots modulo a composite number. In the next section we shall study such one-way functions, before presenting some public key encryption algorithms later in the chapter. However, these are only computational one-way functions in that given enough computing power one can invert these functions faster than exhaustive search.

2. Candidate One-way Functions

The most important one-way function used in public key cryptography is that of factoring integers. By factoring an integer we mean finding its prime factors, for example

$$\begin{aligned} 10 &= 2 \cdot 5 \\ 60 &= 2^2 \cdot 3 \cdot 5 \\ 2^{113} - 1 &= 3391 \cdot 23\,279 \cdot 65\,993 \cdot 1\,868\,569 \cdot 1\,066\,818\,132\,868\,207 \end{aligned}$$

Finding the factors is an expensive computational operation. To measure the complexity of algorithms to factor an integer N we often use the function

$$L_N(\alpha, \beta) = \exp\left((\beta + o(1))(\log N)^{\alpha}(\log\log N)^{1-\alpha}\right).$$

Notice that if an algorithm to factor an integer has complexity $O(L_N(0, \beta))$, then it runs in polynomial time (recall the input size of the problem is $\log N$). However, if an algorithm to factor an integer has complexity $O(L_N(1, \beta))$ then it runs in exponential time. Hence, the function $L_N(\alpha, \beta)$ for $0 < \alpha < 1$ interpolates between polynomial and exponential time. An algorithm with complexity $O(L_N(\alpha, \beta))$ for $0 < \alpha < 1$ is said to have sub-exponential behaviour. Notice that multiplication, which is the inverse algorithm to factoring, is a very simple operation requiring time less than $O(L_N(0, 2))$.

There are a number of methods to factor numbers of the form

$$N = p \cdot q,$$

some of which we shall discuss in a later chapter. For now we just summarize the most well-known techniques.

- **Trial Division:** Try every prime number up to $\sqrt{N}$ and see if it is a factor of N. This has complexity $L_N(1, 1)$, and is therefore an exponential algorithm.
- **Elliptic Curve Method:** This is a very good method if $p < 2^{50}$, its complexity is $L_p(1/2, c)$, which is sub-exponential. Notice that the complexity is given in terms of the size of the unknown value p. If the number is a product of two primes of very unequal size then the elliptic curve method may be the best at finding the factors.
- **Quadratic Sieve:** This is probably the fastest method for factoring integers of between 80 and 100 decimal digits. It has complexity $L_N(1/2, 1)$.

- **Number Field Sieve:** This is currently the most successful method for numbers with more than 100 decimal digits. It can factor numbers of the size of $10^{155} \approx 2^{512}$ and has complexity $L_N(1/3, 1.923)$.

There are a number of other hard problems related to factoring which can be used to produce public key cryptosystems. Suppose you are given N but not its factors p and q, there are four main problems which one can try to solve:

- **FACTORING:** Find p and q.
- **RSA:** Given e such that
$$\gcd(e, (p-1)(q-1)) = 1$$
and c, find m such that
$$m^e = c \pmod{N}.$$
- **QUADRES:** Given a, determine whether a is a square modulo N.
- **SQRROOT:** Given a such that
$$a = x^2 \pmod{N},$$
find x.

Another important class of problems are those based on the discrete logarithm problem or its variants. Let $(G, \cdot)$ be a finite abelian group, such as the multiplicative group of a finite field or the set of points on an elliptic curve over a finite field. The discrete logarithm problem, or DLP, in G is given $g, h \in G$, find an integer x (if it exists) such that
$$g^x = h.$$
For some groups G this problem is easy. For example if we take G to be the integers modulo a number N under addition then given $g, h \in \mathbb{Z}/N\mathbb{Z}$ we need to solve
$$x \cdot g = h.$$
We have already seen in Chapter 1 that we can easily tell whether such an equation has a solution, and determine its solution when it does, using the extended Euclidean algorithm.

For certain other groups determining discrete logarithms is believed to be hard. For example in the multiplicative group of a finite field the best known algorithm for this task is the Number Field Sieve. The complexity of determining discrete logarithms in this case is given by
$$L_N(1/3, c)$$
for some constant c, depending on the type of the finite field, e.g. whether it is a large prime field or an extension field of characteristic two.

For other groups, such as elliptic curve groups, the discrete logarithm problem is believed to be even harder. The best known algorithm for finding discrete logarithms on a general elliptic curve defined over a finite field $\mathbb{F}_q$ is Pollard's Rho method which has complexity
$$\sqrt{q} = L_q(1, 1/2).$$
Hence, this is a fully exponential algorithm. Since determining elliptic curve discrete logarithms is harder than in the case of multiplicative groups of finite fields we are able to use smaller groups. This leads to an advantage in key size. Elliptic curve cryptosystems often have much smaller key sizes (say 160 bits) compared with those based on factoring or discrete logarithms in finite fields (where for both the 'equivalent' recommended key size is about 1024 bits).

Just as with the FACTORING problem, there are a number of related problems associated to discrete logarithms; again suppose we are given a finite abelian group $(G, \cdot)$ and $g \in G$.

- **DLP:** This is the discrete logarithm problem considered above. Namely given $g, h \in G$ such that $h = g^x$, find x.

- **DHP:** This is the Diffie–Hellman problem. Given $g \in G$ and
$$a = g^x \text{ and } b = g^y,$$
find c such that
$$c = g^{xy}.$$
- **DDH:** This is the decision Diffie–Hellman problem. Given $g \in G$ and
$$a = g^x,\ b = g^y \text{ and } c = g^z,$$
determine if $z = x \cdot y$.

When giving all these problems it is important to know how they are all related. This is done by giving complexity theoretic reductions from one problem to another. This allows us to say that 'Problem A is no harder than Problem B'. We do this by assuming an oracle (or efficient algorithm) to solve Problem B. We then use this oracle to give an efficient algorithm for Problem A. Hence, we reduce the problem of solving Problem A to inventing an efficient algorithm to solve Problem B. The algorithms which perform these reductions should be efficient, in that they run in polynomial time, where we treat each oracle query as a single step.

We can also show *equivalence* between two problems A and B, by showing an efficient reduction from A to B and an efficient reduction from B to A. If the two reductions are both polynomial-time reductions then we say that the two problems are *polynomial-time equivalent.*

As an example we first show how to reduce solving the Diffie–Hellman problem to the discrete logarithm problem.

LEMMA 11.1. *In an arbitrary finite abelian group G the DHP is no harder than the DLP.*

PROOF. Suppose I have an oracle $\mathcal{O}_{\text{DLP}}$ which will solve the DLP for me, i.e. on input of $h = g^x$ it will return x. To solve the DHP on input of $a = g^x$ and $b = g^y$ we compute

(1) $z = \mathcal{O}_{\text{DLP}}(a)$.
(2) $c = b^z$.
(3) Output c.

The above reduction clearly runs in polynomial time and will compute the true solution to the DHP, assuming the oracle returns the correct value, i.e.
$$z = x.$$
Hence, the DHP is no harder than the DLP. □

In some groups there is a more complicated argument to show that the DHP is in fact equivalent to the DLP.

We now show how to reduce the solution of the decision Diffie–Hellman problem to the Diffie–Hellman problem, and hence using our previous argument to the discrete logarithm problem.

LEMMA 11.2. *In an arbitrary finite abelian group G the DDH is no harder than the DHP.*

PROOF. Now suppose we have an oracle $\mathcal{O}_{\text{DHP}}$ which on input of g^x and g^y computes the value of g^{xy}. To solve the DDH on input of $a = g^x, b = g^y$ and $c = g^z$ we compute

(1) $d = \mathcal{O}_{\text{DHP}}(a, b)$.
(2) If $d = c$ output YES.
(3) Else output NO.

Again the reduction clearly runs in polynomial time, and assuming the output of the oracle is correct then the above reduction will solve the DDH. □

So the decision Diffie–Hellman problem is no harder than the computational Diffie–Hellman problem. There are however some groups in which one can solve the DDH in polynomial time but the fastest known algorithm to solve the DHP takes sub-exponential time.

Hence, of our three discrete logarithm based problems, the easiest is DDH, then comes DHP and finally the hardest problem is DLP.

We now turn to show reductions for the factoring based problems. The most important result is

LEMMA 11.3. *The FACTORING and SQRROOT problems are polynomial-time equivalent.*

PROOF. We first show how to reduce SQRROOT to FACTORING. Assume we are given a factoring oracle, we wish to show how to use this to extract square roots modulo a composite number N. Namely, given

$$z = x^2 \pmod{N}$$

we wish to compute x. First we factor N into its prime factors p_i using the factoring oracle. Then we compute

$$s_i = \sqrt{z} \pmod{p_i},$$

this can be done in expected polynomial time using Shanks' Algorithm. Then we compute the value of x using the Chinese Remainder Theorem on the data

$$s_i = \sqrt{z} \pmod{p_i}.$$

One has to be a little careful if powers of p_i greater than one divide N, but this is easy to deal with and will not concern us here. Hence, finding square roots modulo N is no harder than factoring.

We now show that FACTORING can be reduced to SQRROOT. Assume we are given an oracle for extracting square roots modulo a composite number N. We shall assume for simplicity that N is a product of two primes, which is the most difficult case. The general case is only slightly more tricky mathematically, but it is computationally easier since factoring numbers with three or more prime factors is usually easier than factoring numbers with two prime factors.

We wish to use our oracle for the problem SQRROOT to factor the integer N into its prime factors, i.e. given $N = p \cdot q$ we wish to compute p. First we pick a random $x \in (\mathbb{Z}/N\mathbb{Z})^*$ and compute

$$z = x^2 \pmod{N}.$$

Now we compute

$$y = \sqrt{z} \pmod{N}$$

using the SQRROOT oracle. There are four such square roots, since N is a product of two primes. With 50 percent probability we obtain

$$y \neq \pm x \pmod{N}.$$

If we do not obtain this inequality then we simply repeat the method. We expect after an average number of two repetitions we will obtain the desired inequality.

Now, since $x^2 = y^2 \pmod{N}$, we see that N divides

$$x^2 - y^2 = (x - y)(x + y).$$

But N does not divide either $x - y$ or $x + y$, since $y \neq \pm x \pmod{N}$. So the factors of N must be distributed over these later two pairs of numbers. This means we can obtain a non-trivial factor of N by computing $\gcd(x - y, N)$

Clearly both of the above reductions can be performed in expected polynomial time. Hence, the problems FACTORING and SQRROOT are polynomial-time equivalent. □

The above proof contains an important tool used in factoring algorithms, namely the construction of a difference of two squares. We shall return to this later in Chapter 12.

Before leaving the problem SQRROOT notice that QUADRES is easier than SQRROOT, since an algorithm to compute square roots modulo N can be used to determine quadratic residuosity.

Finally we end this section by showing that the RSA problem can be reduced to FACTORING. Recall the RSA problem is given $c = m^e \pmod N$, find m.

LEMMA 11.4. *The RSA problem is no harder than the FACTORING problem.*

PROOF. Using a factoring oracle we first find the factorization of N. We can now compute $\Phi = \phi(N)$ and then compute

$$d = 1/e \pmod \Phi.$$

Once d has been computed it is easy to recover m via

$$c^d = m^{ed} = m^{1 \pmod \Phi} = m \pmod N.$$

Hence, the RSA problem is no harder than FACTORING. □

There is some evidence, although slight, that the RSA problem may actually be easier than FACTORING for some problem instances. It is a major open question as to how much easier it is.

3. RSA

The RSA algorithm was the world's first public key encryption algorithm, and it has stood the test of time remarkably well. The RSA algorithm is based on the difficulty of the RSA problem considered in the previous section, and hence it is based on the difficulty of finding the prime factors of large integers. We have seen that it may be possible to solve the RSA problem without factoring, hence the RSA algorithm is not based completely on the difficulty of factoring.

Suppose Alice wishes to enable anyone to send her secret messages, which only she can decrypt. She first picks two large secret prime numbers p and q. Alice then computes

$$N = p \cdot q.$$

Alice also chooses an encryption exponent e which satisfies

$$\gcd(e, (p-1)(q-1)) = 1.$$

It is common to choose $e = 3, 17$ or $65\,537$. Now Alice's public key is the pair (N, e), which she can publish in a public directory. To compute the private key Alice applies the extended Euclidean algorithm to e and $(p-1)(q-1)$ to obtain the decryption exponent d, which should satisfy

$$e \cdot d \equiv 1 \pmod{(p-1)(q-1)}.$$

Alice keeps secret her private key, which is the triple (d, p, q). Actually, she could simply throw away p and q, and retain a copy of her public key which contains the integer N, but we shall see later that this is not efficient.

Now suppose Bob wishes to encrypt a message to Alice. He first looks up Alice's public key and represents the message as a number m which is strictly less than the public modulus N. The ciphertext is then produced by raising the message to the power of the public encryption exponent modulo the public modulus, i.e.

$$c = m^e \pmod N.$$

Alice on receiving c can decrypt the ciphertext to recover the message by exponentiating by the private decryption exponent, i.e.

$$m = c^d \pmod N.$$

This works since the group $(\mathbb{Z}/N\mathbb{Z})^*$ has order

$$\phi(N) = (p-1)(q-1)$$

and so, by Lagrange's Theorem,

$$x^{(p-1)(q-1)} \equiv 1 \pmod{N},$$

for all $x \in (\mathbb{Z}/N\mathbb{Z})^*$. For some integer s we have

$$ed - s(p-1)(q-1) = 1,$$

and so

$$\begin{aligned} c^d &= (m^e)^d \\ &= m^{ed} \\ &= m^{1+s(p-1)(q-1)} \\ &= m \cdot m^{s(p-1)(q-1)} \\ &= m. \end{aligned}$$

To make things clearer let's consider a baby example. Choose $p = 7$ and $q = 11$, and so $N = 77$ and $(p-1)(q-1) = 6 \cdot 10 = 60$. We pick as the public encryption exponent $e = 37$, since we have $\gcd(37, 60) = 1$. Then, applying the extended Euclidean algorithm we obtain $d = 13$ since

$$37 \cdot 13 = 481 = 1 \pmod{60}.$$

Suppose the message we wish to transmit is given by $m = 2$, then to encrypt m we compute

$$c = m^e \pmod{N} = 2^{37} \pmod{77} = 51,$$

whilst to decrypt the ciphertext c we compute

$$m = c^d \pmod{N} = 51^{13} \pmod{77} = 2.$$

3.1. RSA Encryption and the RSA Problem. The security of RSA on first inspection relies on the difficulty of finding the private encryption exponent d given only the public key, namely the public modulus N and the public encryption exponent e.

We have shown that the RSA problem is no harder than FACTORING, hence if we can factor N then we can find p and q and hence we can calculate d. Hence, if factoring is easy we can break RSA. Currently 500-bit numbers are the largest that have been factored and so it is recommended that one takes public moduli of size around 1024 bits to ensure medium-term security. For long-term security one would need to take a public modulus size of over 2048 bits.

In this chapter we shall consider security to be defined as being unable to recover the whole plaintext given the ciphertext. We shall argue in a later chapter that this is far too weak a definition of security for many applications. In addition in a later chapter we shall show that RSA, as we have described it, is not secure against a chosen ciphertext attack.

For a public key algorithm the adversary always has access to the encryption algorithm, hence she can always mount a chosen plaintext attack. RSA is secure against a chosen plaintext attack assuming our weak definition of security and that the RSA problem is hard. To show this we use the reduction arguments of the previous section. This example is rather trivial but we labour the point since these arguments are used over and over again in later chapters.

LEMMA 11.5. *If the RSA problem is hard then the RSA system is secure under a chosen plaintext attack, in the sense that an attacker is unable to recover the whole plaintext given the ciphertext.*

PROOF. We wish to give an algorithm which solves the RSA problem using an algorithm to break the RSA cryptosystem as an oracle. If we can show this then we can conclude that the breaking the RSA cryptosystem is no harder than solving the RSA problem.

Recall that the RSA problem is given $N = p \cdot q$, e and $y \in (\mathbb{Z}/N\mathbb{Z})^*$, compute an x such that $x^e \pmod N = y$. We use our oracle to break the RSA encryption algorithm to 'decrypt' the message corresponding to $c = y$, this oracle will return the plaintext message m. Then our RSA problem is solved by setting $x = m$ since, by definition,

$$m^e \pmod N = c = y.$$

So if we can break the RSA algorithm then we can solve the RSA problem. □

3.2. Knowledge of the Private Exponent and Factoring. Whilst it is unclear whether breaking RSA, in the sense of inverting the RSA function, is equivalent to factoring, determining the private key d given the public information N and e is equivalent to factoring.

LEMMA 11.6. *If one knows the RSA decryption exponent d corresponding to the public key (N, e) then one can efficiently factor N.*

PROOF. Recall that for some integer s

$$ed - 1 = s(p-1)(q-1).$$

We pick an integer $x \neq 0$, this is guaranteed to satisfy

$$x^{ed-1} = 1 \pmod N.$$

We now compute a square root y_1 of one modulo N,

$$y_1 = \sqrt{x^{ed-1}} = x^{(ed-1)/2},$$

which we can do since $ed - 1$ is known and will be even. We will then have the identity

$${y_1}^2 - 1 \equiv 0 \pmod N,$$

which we can use to recover a factor of N via computing

$$\gcd(y_1 - 1, N).$$

But this will only work when $y_1 \neq \pm 1 \pmod N$.

Now suppose we are unlucky and we obtain $y_1 = \pm 1 \pmod N$ rather than a factor of N. If $y_1 = -1 \pmod N$ we return to the beginning and pick another value of x. This leaves us with the case $y_1 = 1 \pmod N$, in which case we take another square root of one via,

$$y_2 = \sqrt{y_1} = x^{(ed-1)/4}.$$

Again we have

$${y_2}^2 - 1 = y_1 - 1 = 0 \pmod N.$$

Hence we compute

$$\gcd(y_2 - 1, N)$$

and see if this gives a factor of N. Again this will give a factor of N unless $y_2 = \pm 1$, if we are unlucky we repeat once more and so on.

This method can be repeated until either we have factored N or until $(ed-1)/2^t$ is no longer divisible by 2. In this latter case we return to the beginning, choose a new random value of x and start again. □

The algorithm in the above proof is an example of a Las Vegas Algorithm: It is probabilistic in nature in the sense that whilst it may not actually give an answer (or terminate), it is however guaranteed that when it does give an answer then that answer will always be correct.

We shall now present a small example of the previous method. Consider the following RSA parameters

$$N = 1\,441\,499, e = 17 \text{ and } d = 507\,905.$$

Recall we are assuming that the private exponent d is public knowledge. We will show that the previous method does in fact find a factor of N. Put

$$\begin{aligned} t_1 &= (ed-1)/2 = 4\,317\,192, \\ x &= 2. \end{aligned}$$

To compute y_1 we evaluate

$$\begin{aligned} y_1 &= x^{(ed-1)/2}, \\ &= 2^{t_1}, \\ &= 1 \pmod{N}. \end{aligned}$$

Since we obtain $y_1 = 1$ we need to set

$$\begin{aligned} t_2 &= t_1/2 = (ed-1)/4 = 2\,158\,596, \\ y_2 &= 2^{t_2}. \end{aligned}$$

We now compute y_2,

$$\begin{aligned} y_2 &= x^{(ed-1)/4}, \\ &= 2^{t_2}, \\ &= 1 \pmod{N}. \end{aligned}$$

So we need to repeat the method again, this time we obtain $t_3 = (ed-1)/8 = 1\,079\,298$. We compute y_3,

$$\begin{aligned} y_3 &= x^{(ed-1)/8}, \\ &= 2^{t_3}, \\ &= 119\,533 \pmod{N}. \end{aligned}$$

So

$$y_3{}^2 - 1 = (y_3 - 1)(y_3 + 1) \equiv 0 \pmod{N},$$

and we compute a prime factor of N by evaluating

$$\gcd(y_3 - 1, N) = 1423.$$

3.3. Knowledge of $\phi(N)$ and Factoring. We have seen that knowledge of d allows us to factor N. Now we will show that knowledge of $\Phi = \phi(N)$ also allows us to factor N.

LEMMA 11.7. *Given an RSA modulus N and the value of $\Phi = \phi(N)$ one can efficiently factor N.*

PROOF. We have

$$\Phi = (p-1)(q-1) = N - (p+q) + 1.$$

Hence, if we set $S = N + 1 - \Phi$, we obtain

$$S = p + q.$$

So we need to determine p and q from their sum S and product N. Define the polynomial

$$f(X) = (X - p) \cdot (X - q) = X^2 - SX + N.$$

So we can find p and q by solving $f(X) = 0$ using the standard formulae for extracting the roots of a quadratic polynomial,

$$p = \frac{S + \sqrt{S^2 - 4N}}{2},$$
$$q = \frac{S - \sqrt{S^2 - 4N}}{2}.$$

□

As an example consider the RSA public modulus $N = 18\,923$. Assume that we are given $\Phi = \phi(N) = 18\,648$. We then compute

$$S = p + q = N + 1 - \Phi = 276.$$

Using this we compute the polynomial

$$f(X) = X^2 - SX + N = X^2 - 276X + 18\,923$$

and find that its roots over the real numbers are

$$p = 149, q = 127$$

which are indeed the factors of N.

3.4. Use of a Shared Modulus. Since modular arithmetic is very expensive it can be very tempting for a system to be set up in which a number of users share the same public modulus N but use different public/private exponents, (e_i, d_i). One reason to do this could be to allow very fast hardware acceleration of modular arithmetic, specially tuned to the chosen shared modulus N. This is, however, a very silly idea since it can be attacked in one of two ways, either by a malicious insider or by an external attacker.

Suppose the bad guy is one of the internal users, say user number one. He can now compute the value of the decryption exponent for user number two, namely d_2. First user one computes p and q since they know d_1, via the algorithm in the proof of Lemma 11.6. Then user one computes $\phi(N) = (p-1)(q-1)$, and finally they can recover d_2 from

$$d_2 = \frac{1}{e_2} \pmod{\phi(N)}.$$

Now suppose the attacker is not one of the people who share the modulus. Suppose Alice sends the same message m to two of the users with public keys

$$(N, e_1) \text{ and } (N, e_2),$$

i.e. $N_1 = N_2 = N$. Eve, the external attacker, sees the messages c_1 and c_2 where

$$c_1 = m^{e_1} \pmod{N},$$
$$c_2 = m^{e_2} \pmod{N}.$$

Eve can now compute

$$t_1 = {e_1}^{-1} \pmod{e_2},$$
$$t_2 = (t_1 e_1 - 1)/e_2,$$

and can recover the message m from

$$\begin{aligned} c_1^{t_1} c_2^{-t_2} &= m^{e_1 t_1} m^{-e_2 t_2} \\ &= m^{1+e_2 t_2} m^{-e_2 t_2} \\ &= m^{1+e_2 t_2 - e_2 t_2} \\ &= m^1 = m. \end{aligned}$$

As an example of this external attack, take the public keys as

$$N = N_1 = N_2 = 18\,923,\ e_1 = 11 \text{ and } e_2 = 5.$$

Now suppose Eve sees the ciphertexts

$$c_1 = 1514 \text{ and } c_2 = 8189$$

corresponding to the same plaintext m. Then Eve computes $t_1 = 1$ and $t_2 = 2$, and recovers the message

$$m = c_1^{t_1} c_2^{-t_2} = 100 \pmod{N}.$$

3.5. Use of a Small Public Exponent. Fielded RSA systems often use a small public exponent e so as to cut down the computational cost of the sender. We shall now show that this can also lead to problems. Suppose we have three users all with different public moduli

$$N_1,\ N_2 \text{ and } N_3.$$

In addition suppose they all have the same small public exponent $e = 3$. Suppose someone sends them the same message m.

The attacker Eve sees the messages

$$\begin{aligned} c_1 &= m^3 \pmod{N_1}, \\ c_2 &= m^3 \pmod{N_2}, \\ c_3 &= m^3 \pmod{N_3}. \end{aligned}$$

Now the attacker, using the Chinese Remainder Theorem, computes the simultaneous solution to

$$X = c_i \pmod{N_i} \text{ for } i = 1, 2, 3,$$

to obtain

$$X = m^3 \pmod{N_1 N_2 N_3}.$$

But since $m^3 < N_1 N_2 N_3$ we must have $X = m^3$ identically over the integers. Hence we can recover m by taking the real cube root of X.

As a simple example of this attack take,

$$N_1 = 323,\ N_2 = 299 \text{ and } N_3 = 341.$$

Suppose Eve sees the ciphertexts

$$c_1 = 50,\ c_2 = 268 \text{ and } c_3 = 1,$$

and wants to determine the common value of m. Eve computes via the Chinese Remainder Theorem

$$X = 300\,763 \pmod{N_1 N_2 N_3}.$$

Finally, she computes over the integers

$$m = X^{1/3} = 67.$$

This attack and the previous one are interesting since we find the message without factoring the modulus. This is, albeit slight, evidence that breaking RSA is easier than factoring. The main lesson, however, from both these attacks is that plaintext should be randomly padded before transmission. That way the same 'message' is never encrypted to two different people. In addition

one should probably avoid very small exponents for encryption, $e = 65\,537$ is the usual choice now in use. However, small public exponents for RSA signatures (see later) do not seem to have any problems.

4. ElGamal Encryption

The simplest encryption algorithm based on the discrete logarithm problem is the ElGamal encryption algorithm. In the following we shall describe the finite field analogue of ElGamal encryption, we leave it as an exercise to write down the elliptic curve variant.

Unlike the RSA algorithm, in ElGamal encryption there are some public parameters which can be shared by a number of users. These are called the domain parameters and are given by

- p a 'large prime', by which we mean one with around 1024 bits, such that $p-1$ is divisible by another 'medium prime' q of around 160 bits.
- g an element of $\mathbb{F}_p^*$ of prime order q, i.e.

$$g = r^{(p-1)/q} \pmod{p} \neq 1 \text{ for some } r \in \mathbb{F}_p^*.$$

All the domain parameters do is create a public finite abelian group G of prime order q with generator g. Such domain parameters can be shared between a large number of users.

Once these domain parameters have been fixed, the public and private keys can then be determined. The private key is chosen to be an integer x, whilst the public key is given by

$$h = g^x \pmod{p}.$$

Notice that whilst each user in RSA needed to generate two large primes to set up their key pair (which is a costly task), for ElGamal encryption each user only needs to generate a random number and perform a modular exponentiation to generate a key pair.

Messages are assumed to be non-zero elements of the field $\mathbb{F}_p^*$. To encrypt a message $m \in \mathbb{F}_p^*$ we

- generate a random ephemeral key k,
- set $c_1 = g^k$,
- set $c_2 = m \cdot h^k$,
- output the ciphertext as $c = (c_1, c_2)$.

Notice that since each message has a different ephemeral key, encrypting the same message twice will produce different ciphertexts.

To decrypt a ciphertext $c = (c_1, c_2)$ we compute

$$\begin{aligned} \frac{c_2}{{c_1}^x} &= \frac{m \cdot h^k}{g^{xk}} \\ &= \frac{m \cdot g^{xk}}{g^{xk}} \\ &= m. \end{aligned}$$

As an example of ElGamal encryption consider the following. We first need to set up the domain parameters. For our small example we choose

$$q = 101,\ p = 809 \text{ and } g = 3.$$

Note that q divides $p-1$ and that g has order divisible by q in the multiplicative group of integers modulo p. The actual order of g is 808 since

$$3^{808} = 1 \pmod{p},$$

and no smaller power of g is equal to one. As a public private key pair we choose

- $x = 68$,
- $h = g^x = 65$.

Now suppose we wish to encrypt the message $m = 100$ to the user with the above ElGamal public key.

- We generate a random ephemeral key $k = 89$.
- Set $c_1 = g^k = 345$.
- Set $c_2 = m \cdot h^k = 517$.
- Output the ciphertext as $c = (345, 517)$.

The recipient can decrypt our ciphertext by computing

$$\begin{aligned} \frac{c_2}{{c_1}^x} &= \frac{517}{345^{68}} \\ &= 100. \end{aligned}$$

This last value is computed by first computing 345^{68}, taking the inverse modulo p of the result and then multiplying this value by 517.

In a later chapter we shall see that ElGamal encryption as it stands is not secure against a chosen ciphertext attack, so usually a modified scheme is used. However, ElGamal encryption is secure against a chosen plaintext attack, assuming the Diffie–Hellman problem is hard. Again, here we take a naive definition of what security means in that an encryption algorithm is secure if an adversary is unable to invert the encryption function.

LEMMA 11.8. *Assuming the Diffie–Hellman problem (DHP) is hard then ElGamal is secure under a chosen plaintext attack, where security means it is hard for the adversary, given the ciphertext, to recover the whole of the plaintext.*

PROOF. To see that ElGamal encryption is secure under a chosen plaintext attack assuming the Diffie–Hellman problem is hard, we first suppose that we have an oracle $\mathcal{O}$ to break ElGamal encryption. This oracle $\mathcal{O}(h, (c_1, c_2))$ takes as input a public key h and a ciphertext (c_1, c_2) and then returns the underlying plaintext. We will then show how to use this oracle to solve the DHP.

Suppose we are given

$$g^x \text{ and } g^y$$

and we are asked to solve the DHP, i.e. we need to compute g^{xy}.

We first set up an ElGamal public key which depends on the input to this Diffie–Hellman problem, i.e. we set

$$h = g^x.$$

Note, we do not know what the corresponding private key is. Now we write down the 'ciphertext'

$$c = (c_1, c_2),$$

where

- $c_1 = g^y$,
- c_2 is a random element of $\mathbb{F}_p^*$.

Now we input this ciphertext into our oracle which breaks ElGamal encryption so as to produce the corresponding plaintext, $m = \mathcal{O}(h, (c_1, c_2))$. We can now solve the original Diffie–Hellman problem by computing

$$\begin{aligned} \frac{c_2}{m} &= \frac{m \cdot h^y}{m} \text{ since } c_1 = g^y \\ &= h^y \\ &= g^{xy}. \end{aligned}$$

□

5. Rabin Encryption

There is another system, due to Rabin, based on the difficulty of factoring large integers. In fact it is actually based on the difficulty of extracting square roots modulo $N = p \cdot q$. Recall that these two problems are known to be equivalent, i.e.

- knowing the factors of N means we can extract square roots modulo N,
- extracting square roots modulo N means we can factor N.

Hence, in some respects such a system should be considered more secure than RSA. Encryption in the Rabin encryption system is also much faster than almost any other public key scheme. Despite these plus points the Rabin system is not used as much as the RSA system. It is, however, useful to study for a number of reasons, both historical and theoretical. The basic idea of the system is also used in some higher level protocols.

We first choose prime numbers of the form

$$p \equiv q \equiv 3 \pmod 4$$

since this makes extracting square roots modulo p and q very fast. The private key is then the pair (p, q). To compute the associated public key we generate a random integer $B \in \{0, \ldots, N-1\}$ and then the public key is

$$(N, B),$$

where N is the product of p and q.

To encrypt a message m, using the above public key, in the Rabin encryption algorithm we compute

$$c = m(m + B) \pmod N.$$

Hence, encryption involves one addition and one multiplication modulo N. Encryption is therefore much faster than RSA encryption, even when one chooses a small RSA encryption exponent.

Decryption is far more complicated, essentially we want to compute

$$m = \sqrt{\frac{B^2}{4} + c} - \frac{B}{2} \pmod N.$$

At first sight this uses no private information, but a moment's thought reveals that you need the factorization of N to be able to find the square root. There are however four possible square roots modulo N, since N is the product of two primes. Hence, on decryption you obtain four possible plaintexts. This means that we need to add redundancy to the plaintext before encryption in order to decide which of the four possible plaintexts corresponds to the intended one.

We still need to show why Rabin decryption works. Recall

$$c = m(m + B) \pmod N,$$

then

$$\begin{aligned}
\sqrt{\frac{B^2}{4} + c} - \frac{B}{2} &= \sqrt{\frac{B^2 + 4m(m+B)}{4}} - \frac{B}{2} \\
&= \sqrt{\frac{4m^2 + 4Bm + B^2}{4}} - \frac{B}{2} \\
&= \sqrt{\frac{(2m+B)^2}{4}} - \frac{B}{2} \\
&= \frac{2m + B}{2} - \frac{B}{2} \\
&= m,
\end{aligned}$$

of course assuming the 'correct' square root is taken.

We end with an example of Rabin encryption at work. Let the public and private keys be given by

- $p = 127$ and $q = 131$,
- $N = 16\,637$ and $B = 12\,345$.

To encrypt $m = 4410$ we compute

$$c = m(m + B) \pmod{N} = 4633.$$

To decrypt we first compute

$$t = B^2/4 + c \pmod{N} = 1500.$$

We then evaluate the square root of t modulo p and q

$$\sqrt{t} \pmod{p} = \pm 22,$$
$$\sqrt{t} \pmod{q} = \pm 37.$$

Now we apply the Chinese Remainder Theorem to both

$$\pm 22 \pmod{p} \text{ and } \pm 37 \pmod{q}$$

so as to find the square root of t modulo N,

$$s = \sqrt{t} \pmod{N} = \pm 3705 \text{ or } \pm 14\,373.$$

The four possible messages are then given by the four possible values of

$$s - \frac{B}{2} = s - \frac{12\,345}{2}.$$

This leaves us with the four messages

$$4410, 5851, 15\,078, \text{ or } 16\,519.$$

Chapter Summary

- Public key encryption requires one-way functions. Examples of these are FACTORING, SQRROOT, DLP, DHP and DDH.
- There are a number of relationships between these problems. These relationships are proved by assuming an oracle for one problem and then using this in an algorithm to solve the other problem.
- RSA is the most popular public key encryption algorithm, but its security rests on the difficulty of the RSA problem and not quite on the difficulty of FACTORING.
- ElGamal encryption is a system based on the difficulty of the Diffie–Hellman problem (DHP).
- Rabin encryption is based on the difficulty of extracting square roots modulo a composite modulus. Since the problems SQRROOT and FACTORING are polynomial-time equivalent this means that Rabin encryption is based on the difficulty of FACTORING.

Further Reading

Still the best quick introduction to the concept of public key cryptography can be found in the original paper of Diffie and Hellman. See also the original papers on ElGamal, Rabin and RSA encryption.

W. Diffie and M. Hellman. *New directions in cryptography.* IEEE Trans. on Info. Theory, **22**, 644–654, 1976.

T. ElGamal. *A public key cryptosystem and a signature scheme based on discrete logarithms.* IEEE Trans. Info. Theory, **31**, 469–472, 1985.

R.L. Rivest, A. Shamir and L.M. Adleman. *A method for obtaining digital signatures and public-key cryptosystems.* Comm. ACM, **21**, 120–126, 1978.

M. Rabin. *Digitized signatures and public key functions as intractable as factorization.* MIT/LCS/TR-212, MIT Laboratory for Computer Science, 1979.

CHAPTER 12

Primality Testing and Factoring

Chapter Goals

- To explain the basics of primality testing.
- To describe the most used primality testing algorithm, namely Miller–Rabin.
- To explain various factoring algorithms.
- To sketch how the most successful factoring algorithm works, namely the Number Field Sieve.

1. Prime Numbers

The generation of prime numbers is needed for almost all public key algorithms, for example

- In the RSA or the Rabin system we need to find primes p and q to compute the public key $N = p \cdot q$.
- In ElGamal encryption we need to find p and q with q dividing $p - 1$.
- In the elliptic curve variant of ElGamal we require an elliptic curve over a finite field, such that the order of the elliptic curve is divisible by a large prime q.

Luckily we shall see that testing a number for primality can be done very fast using very simple code, but with an algorithm which has a probability of error. By repeating this algorithm we can reduce the error probability to any value that we require.

Some of the more advanced primality testing techniques will produce a certificate which can be checked by a third party to prove that the number is indeed prime. Clearly one requirement of such a certificate is that it should be quicker to verify than it is to generate. Such a primality testing routine will be called a primality proving algorithm, and the certificate will be called a proof of primality. However, the main primality testing algorithm used in cryptographic systems only produces certificates of compositeness and not certificates of primality.

Before discussing these algorithms we need to look at some basic heuristics concerning prime numbers. A famous result in mathematics, conjectured by Gauss after extensive calculation in the early 1800s, is the Prime Number Theorem:

THEOREM 12.1 (Prime Number Theorem). *The function $\pi(X)$ counts the number of primes less than X, where we have the approximation*

$$\pi(X) \approx \frac{X}{\log X}.$$

This means primes are quite common. For example the number of primes less than 2^{512} is about 2^{503}.

The Prime Number Theorem also allows us to estimate what the probability of a random number being prime is: if p is a number chosen at random then the probability it is prime is about

$$\frac{1}{\log p}.$$

So a random number p of 512 bits in length will be a prime with probability

$$\approx \frac{1}{\log p} \approx \frac{1}{355}.$$

So on average we need to select 177 odd numbers of size 2^{512} before we find one which is prime. Hence, it is practical to generate large primes, as long as we can test primality efficiently.

1.1. Trial Division. The naive test for testing a number p to be prime is one of trial division. We essentially take all numbers between 2 and $\sqrt{p}$ and see if they divide p, if not then p is prime. If such a number does divide p then we obtain the added bonus of finding a factor of the composite number p. Hence, trial division has the advantage (compared with more advanced primality testing/proving algorithms) that it either determines that p is a prime, or determines a non-trivial factor of p.

However, primality testing by using trial division is a terrible strategy. In the worst case, when p is a prime, the algorithm requires $\sqrt{p}$ steps to run, which is an exponential function in terms of the size of the input to the problem. Another drawback is that it does not produce a certificate for the primality of p, in the case when the input p is prime. When p is not prime it produces a certificate which can easily be checked to prove that p is composite, namely a non-trivial factor of p. But when p is prime the only way we can verify this fact again (say to convince a third party) is to repeat the algorithm once more.

Despite its drawbacks trial division is however the method of choice for numbers which are very small. In addition partial trial division, up to a bound Y, is able to eliminate all but a proportion

$$\prod_{p<Y} \left(1 - \frac{1}{p}\right)$$

of all composites. Naively this is what one would always do, since for example one would never check an even number greater than two for primality since it is obviously composite. Hence, many primality testing algorithms first do trial division with all primes up to say 100, so as to eliminate all but

$$\prod_{p<100} \left(1 - \frac{1}{p}\right) \approx 0.12$$

of composites.

1.2. Fermat's Test. Most advanced probabilistic algorithms for testing primality make use of the converse to Fermat's Little Theorem. Recall, if G is a multiplicative group of size $\#G$ then

$$a^{\#G} = 1$$

for a random $a \in G$, which was Lagrange's Theorem. So if G is the group of integers modulo n under multiplication then

$$a^{\phi(n)} = 1 \pmod{n}$$

for all $a \in (\mathbb{Z}/n\mathbb{Z})^*$. Fermat's Little Theorem was the case where $n = p$ is prime, in which case the above equality becomes

$$a^{p-1} = 1 \pmod{p}.$$

So if n is prime we have that

$$a^{n-1} = 1 \pmod{n}$$

always holds, whilst if n is not prime then we have that

$$a^{n-1} = 1 \pmod{n}$$

is unlikely to hold.

Since computing $a^{n-1} \pmod{n}$ is a very fast operation this gives us a very fast test for compositeness, called the Fermat test to the base a. Note, running the Fermat test can only convince us of the compositeness of n, it can never prove to us that a number is prime, only that it is not prime.

To see how it does not prove primality consider the case $n = 11 \cdot 31 = 341$ and the base $a = 2$, we have

$$a^{n-1} = 2^{340} = 1 \pmod{341}.$$

but n is clearly not prime. In such a case we say that n is a (Fermat) pseudo-prime to the base 2. There are infinitely many pseudo-primes to any given base, although the pseudo-primes are in fact rarer than the primes. It can be shown that if n is composite then, with probability greater than $1/2$, we obtain

$$a^{n-1} \neq 1 \pmod{n}.$$

This gives us Algorithm 12.1 to test n for primality

Algorithm 12.1: Fermat's test for primality

for $i = 0$ **to** $k - 1$ **do**
  Pick a from $[2, ..., n-1]$
  $b = a^{n-1} \mod n$
  if $b \neq 1$ **then return** (Composite, a)
end
return ("Probably Prime")

If the above outputs (Composite,a) then we know

- n is definitely a composite number,
- a is a witness for this compositeness, in that one can verify that n is composite by using the value of a.

If the above algorithm outputs "Probably Prime" then

- n is a composite with probability at most $1/2^k$,
- n is either a prime or a so-called probable prime.

For example if we take

$$n = 43\,040\,357$$

then n is clearly a composite, with one witness given by $a = 2$ since

$$2^{n-1} \pmod{n} = 9\,888\,212.$$

As another example if we take

$$n = 2^{192} - 2^{64} - 1$$

then the algorithm outputs "Probably Prime" since we cannot find a witness for compositeness. Actually this n is a prime, so it is not surprising we did not find a witness.

However, there are composite numbers for which the Fermat test will output

"Probably Prime"

for every a coprime to n. These numbers are called Carmichael numbers, and to make things worse there are infinitely many of them. The first three are given by $561, 1105$ and 1729. Carmichael numbers have the following properties

- They are always odd.

- They have at least three prime factors.
- They are square free.
- If p divides a Carmichael number N then $p-1$ divides $N-1$.

To give you some idea of their density, if we look at all numbers less than

$$10^{16}$$

then there are about

$$2.7 \cdot 10^{14}$$

primes in this region, but only

$$246\,683 \approx 2.4 \cdot 10^5$$

Carmichael numbers in this region. Hence, Carmichael numbers are rare, but not rare enough to be ignored completely.

1.3. Miller–Rabin Test. Due to the existence of Carmichael numbers the Fermat test is usually avoided. However, there is a modification of the Fermat test, called the Miller–Rabin test, which avoids the problem of composites for which no witness exists. This does not mean it is easy to find a witness for each composite, it only means that a witness must exist. In addition the Miller–Rabin test has probability of 1/4 of accepting a composite as prime for each random base a, so again repeated application of the algorithm leads us to reduce the error probability down to any value we care to mention.

The Miller–Rabin test is given by the pseudo-code in Algorithm 12.2 We do not show that the Miller–Rabin test works. If you are interested in the reason see any book on algorithmic number theory for the details, for example that by Cohen or Bach and Shallit mentioned in the Further Reading section of this chapter. Just as with the Fermat test we repeat the method k times with k different bases, to obtain an error probability of $1/4^k$ if the algorithm always returns "Probably Prime". Hence, we expect that the Miller–Rabin test will output "Probably Prime" for values of $k \geq 20$ only when n is actually a prime.

Algorithm 12.2: Miller–Rabin Algorithm

```
Write n − 1 = 2^s m, with m odd
for j = 0 to k − 1 do
    pick a from [2, ..., n − 2]
    b = a^m mod n
    if b ≠ 1 and b ≠ (n − 1) then
        i = 1  while i < s and b ≠ (n − 1) do
            b = b^2 mod n
            if b = 1 then return (Composite, a)
            i = i + 1
        end
        if b ≠ (n − 1) then return (Composite, a)
    end
end
return ("Probable Prime")
```

If n is a composite then a is called a Miller–Rabin witness for the compositeness of n, and under the Generalized Riemann Hypothesis (GRH), a conjecture believed to be true by most mathematicians, there is a Miller–Rabin witness a for the compositeness of n with

$$a \leq O((\log n)^2).$$

1.4. Primality Proofs. Up to now we have only output witnesses for compositeness, and we can interpret such a witness as a proof of compositeness. In addition we have only obtained probable primes rather than numbers which are 100 percent guaranteed to be prime. In practice this seems to be all right, since the probability of a composite number passing the Miller–Rabin test for twenty bases is around 2^{-40} which should never really occur in practice. But theoretically (and maybe in practice if we are totally paranoid) this could be a problem. In other words we may want real primes and not just probable ones.

There are algorithms whose output is a witness for the primality of the number. Such a witness is called a proof of primality. In practice such programs are only called when one is morally certain that the number one is testing for primality is actually prime. In other words the number has already passed the Miller–Rabin test for a number of bases and all one now requires is a proof of the primality.

The most successful of these primality proving algorithms is one based on elliptic curves called ECPP (for Elliptic Curve Primality Prover). This itself is based on an older primality proving algorithm based on finite fields due to Pocklington and Lehmer, the elliptic curve variant is due to Goldwasser and Kilian. The ECPP algorithm is a randomized algorithm which is not mathematically guaranteed to always produce an output, i.e. a witness, when the input is a prime number. If the input is composite then the algorithm is not guaranteed to terminate at all. Although ECPP runs in polynomial time, i.e. it is quite efficient, the proofs of primality it produces can be verified even faster.

There is an algorithm due to Adleman and Huang which, unlike the ECPP method, is guaranteed to terminate with a proof of primality on input of a prime number. It is based on a generalization of elliptic curves called hyperelliptic curves and has never to my knowledge been implemented. The fact that it has never been implemented is not only due to the far more complicated mathematics involved, but is also due to the fact that while the hyperelliptic variant is mathematically guaranteed to produce a proof, the ECPP method will always do so in practice for less work effort.

2. Factoring Algorithms

Factoring methods are usually divided into Dark Age methods such as

- Trial division,
- $p-1$ method,
- $p+1$ method,
- Pollard rho method,

and modern methods

- Continued Fraction Method (CFRAC),
- Quadratic Sieve (QS),
- Elliptic Curve Method (ECM),
- Number Field Sieve (NFS).

We do not have the time or space to discuss all of these in detail so we shall look at a couple of Dark Age methods and explain the main ideas behind some of the modern methods.

Modern factoring algorithms lie somewhere between polynomial and exponential time, in an area called sub-exponential time. These algorithms have complexity measured by the function

$$L_N(\alpha,\beta) = \exp\left((\beta+o(1))(\log N)^{\alpha}(\log\log N)^{1-\alpha}\right).$$

Notice that

- $L_N(0,\beta) = (\log N)^{\beta+o(1)}$, i.e. polynomial time,
- $L_N(1,\beta) = N^{\beta+o(1)}$, i.e. exponential time.

So, as we remarked in Chapter 11, in some sense the function $L_N(\alpha, \beta)$ interpolates between polynomial and exponential time. To give some idea of the use of this function:

- The slowest factoring algorithm, trial division, has complexity
$$L_N(1, 1/2).$$
- Up to the early 1990s the fastest general purpose factoring algorithm was the Quadratic Sieve, which has complexity
$$L_N(1/2, c)$$
for some constant c.
- The current fastest algorithm is the Number Field Sieve which has complexity
$$L_N(1/3, c)$$
for some constant c.

2.1. Trial Division. The most elementary algorithm is trial division, which we have already met in the context of testing primality. Suppose N is the number we wish to factor, we proceed as described in Algorithm 12.3.

Algorithm 12.3: Factoring via trial division

```
for p = 2 to √N do
    e = 0
    if (N mod p) = 0 then
        while (N mod p) = 0 do
            e = e + 1
            N = N/p
        end
        output (p, e)
    end
end
```

A moment's thought reveals that trial division takes time at worst
$$O(\sqrt{N}).$$
The input size to the algorithm is of size $\log_2 N$, hence this complexity is exponential. But just as in primality testing we should not ignore trial division, it is usually the method of choice for numbers less than 10^{12}.

2.2. Smooth Numbers. For larger numbers we would like to improve on the trial division algorithm. Almost all other factoring algorithms make use of other auxiliary numbers called smooth numbers. Essentially a smooth number is one which is easy to factor using trial division, the following definition makes this more precise.

DEFINITION 12.2 (Smooth Number). *Let B be an integer. An integer N is called B-smooth if every prime factor p of N is less than B.*

For example
$$N = 2^{78} \cdot 3^{89} \cdot 11^3$$
is 12-smooth. Sometimes we say that the number is just smooth if the bound B is small compared with N.

The number of y-smooth numbers which are less than x is given by the function $\psi(x, y)$. This is a rather complicated function which is approximated by
$$\psi(x, y) \approx x\rho(u)$$

where ρ is the Dickman–de Bruijn function and

$$u = \frac{\log x}{\log y}.$$

The Dickman–de Bruijn function ρ is defined as the function which satisfies the following differential-delay equation

$$u\rho'(u) + \rho(u-1) = 0$$

for $u > 1$. In practice we approximate $\rho(u)$ via the expression

$$\rho(u) \approx u^{-u}$$

which holds as $u \to \infty$. This leads to the following result, which is important in analysing advanced factoring algorithms.

THEOREM 12.3. *The proportion of integers less than x, which are $x^{1/u}$-smooth, is asymptotically equal to u^{-u}.*

Now if we set

$$y = L_N(\alpha, \beta)$$

then

$$\begin{aligned} u &= \frac{\log N}{\log y} \\ &= \frac{1}{\beta}\left(\frac{\log N}{\log\log N}\right)^{1-\alpha}. \end{aligned}$$

Hence, one can show

$$\begin{aligned} \frac{1}{N}\psi(N, y) &\approx u^{-u} \\ &= \exp(-u\log u) \\ &\approx \frac{1}{L_N(1-\alpha, \gamma)}, \end{aligned}$$

for some constant γ. Suppose we are looking for numbers less than N which are $L_N(\alpha, \beta)$-smooth. The probability that any number less than N is actually $L_N(\alpha, \beta)$-smooth is given by $1/L_N(1-\alpha, \gamma)$. This hopefully will explain intuitively why some of the modern method complexity estimates for factoring are around $L_N(0.5, c)$, since to balance the smoothness bound against the probability estimate we take $\alpha = \frac{1}{2}$. The number field sieve obtains a better complexity estimate only by using a more mathematically complex algorithm.

We shall also require, in discussing our next factoring algorithm, the notion of a number being B-power smooth:

DEFINITION 12.4 (Power Smooth). *A number is said to be B-power smooth if every prime power dividing N is less than B.*

For example

$$N = 2^5 \cdot 3^3$$

is 33-power smooth.

2.3. Pollard's $P-1$ Method. The most famous name in factoring algorithms in the late 20th century was John Pollard. Almost all the important advances in factoring were made by him, for example

- the $P-1$ method,
- the Rho-method,
- the Number Field Sieve.

In this section we discuss the $P-1$ method and in a later section we consider the Number Field Sieve method, the other methods we leave for the exercises at the end of the chapter.

Suppose the number we wish to factor is given by

$$N = p \cdot q.$$

In addition suppose we know (by some pure guess) an integer B such that $p-1$ is B-power smooth, but that $q-1$ is not B-power smooth. We can then hope that $p-1$ divides $B!$, but that $q-1$ is unlikely to divide $B!$.

Suppose that we compute

$$a = 2^{B!} \pmod{N}.$$

Imagine that we could compute this modulo p and modulo q, we would then have

$$a \equiv 1 \pmod{p},$$

since

- $p-1$ divides $B!$,
- $a^{p-1} = 1 \pmod{p}$ by Fermat's Little Theorem.

But it is unlikely that we would have

$$a \equiv 1 \pmod{q}.$$

Hence,

- p will divide $a-1$,
- q will not divide $a-1$.

We can then recover p by computing

$$p = \gcd(a-1, N),$$

as in Algorithm 12.4

Algorithm 12.4: Pollard's $P-1$ factoring method

```
a = 2
for j = 2 to B do
    a = a^j  mod N
end
p = gcd(a − 1, N)
if p ≠ 1 and p ≠ N then return ("p is a factor of N")
else return ("No Result")
```

As an example, suppose we wish to factor

$$N = 15\,770\,708\,441.$$

We take $B = 180$ and running the above algorithm we obtain

$$a = 2^{B!} \pmod{N} = 1\,162\,022\,425.$$

Then we obtain

$$p = \gcd(a-1, N) = 135\,979.$$

To see why this works in this example we see that the prime factorization of N is given by

$$N = 135\,979 \cdot 115\,979$$

and we have

$$p - 1 = 135\,978 - 1 = 2 \cdot 3 \cdot 131 \cdot 173,$$
$$q - 1 = 115\,978 - 1 = 2 \cdot 103 \cdot 563.$$

Hence $p - 1$ is indeed B-power smooth, whilst $q - 1$ is not B-power smooth.

One can show that the complexity of the $P - 1$ method is given by

$$O(B \log B (\log N)^2 + (\log N)^3).$$

So if we choose $B = O((\log N)^i)$, for some integer i, then this is a polynomial-time factoring algorithm, but it only works for numbers of a special form.

Due to the $P - 1$ method one often sees recommended that RSA primes are chosen to satisfy

$$p - 1 = 2p_1 \text{ and } q - 1 = 2q_1,$$

where p_1 and q_1 are both primes. In this situation the primes p and q are called safe primes. However, this is not really needed these days with the large RSA moduli we use in current applications. This is because the probability that for a random 512-bit prime p, the number $p - 1$ is B-power smooth for a small value of B is very small. Hence, choosing random 512-bit primes would in all likelihood render the $P - 1$ method useless.

2.4. Difference of Two Squares. A basic trick in factoring algorithms, known for many centuries, is to produce two numbers x and y, of around the same size as N, such that

$$x^2 = y^2 \pmod{N}.$$

Since then we have

$$x^2 - y^2 = (x - y)(x + y) = 0 \pmod{N}.$$

If $N = p \cdot q$ then we have four possible cases

(1) p divides $x - y$ and q divides $x + y$.
(2) p divides $x + y$ and q divides $x - y$.
(3) p and q both divide $x - y$ but neither divide $x + y$.
(4) p and q both divide $x + y$ but neither divide $x - y$.

All these cases can occur with equal probability, namely $\frac{1}{4}$. If we then compute

$$d = \gcd(x - y, N),$$

our previous four cases then divide into the cases

(1) $d = p$.
(2) $d = q$.
(3) $d = N$.
(4) $d = 1$.

Since all these cases occur with equal probability, we see that with probability $\frac{1}{2}$ we will obtain a non-trivial factor of N. The only problem is, how do we find x and y such that $x^2 = y^2 \pmod{N}$?

3. Modern Factoring Methods

Most modern factoring methods have the following strategy based on the difference of two squares method described at the end of the last section.

- Take a smoothness bound B.
- Compute a *factorbase* F of all prime numbers p less than B.
- Find a large number of values of x and y, such that x and y are B-smooth and
$$x = y \pmod{N}.$$
These are called *relations* on the factorbase.
- Using linear algebra modulo 2, find a combination of the relations to give an X and Y with
$$X^2 = Y^2 \pmod{N}.$$
- Attempt to factor N by computing $\gcd(X - Y, N)$.

The trick in all algorithms of this form is how to find the relations. All the other details of the algorithms are basically the same. Such a strategy can also be used to solve discrete logarithm problems as well, which we shall discuss in a later chapter. In this section we explain the parts of the modern factoring algorithms which are common and justify why they work.

One way of looking at such algorithms is in the context of computational group theory. We have already shown that knowing the order of the group $(\mathbb{Z}/N\mathbb{Z})^*$, for an RSA modulus N, is the same as knowing the prime factors of N. Hence, the problem is really one of computing a group order. The factorbase is essentially a set of generators of the group $(\mathbb{Z}/N\mathbb{Z})^*$, whilst the relations are relations between the generators of this group. Once a sufficiently large number of relations have been found then, since the group is a finite abelian group, standard group theoretic algorithms will compute the group structure and hence the group order. These general group theoretic algorithms could include computing the Smith Normal Form of the associated matrix. Hence, it should not be surprising that linear algebra is used on the relations so as to factor the integer N.

3.1. Combining Relations. The Smith Normal Form algorithm is far too complicated for factoring algorithms where a more elementary approach, still based on linear algebra, can be used, as we shall now explain. Suppose we have the relations

$$\begin{aligned} p^2q^5r^2 &= p^3q^4r^3 \pmod{N}, \\ pq^3r^5 &= pqr^2 \pmod{N}, \\ p^3q^5r^3 &= pq^3r^2 \pmod{N}, \end{aligned}$$

where p, q and r are primes in our factorbase, $F = \{p, q, r\}$. Dividing one side by the other in each of our relations we obtain

$$\begin{aligned} p^{-1}qr^{-1} &= 1 \pmod{N}, \\ q^2r^3 &= 1 \pmod{N}, \\ p^2q^2r &= 1 \pmod{N}. \end{aligned}$$

Multiplying the last two equations together we obtain

$$p^{0+2}q^{2+2}r^{3+1} \equiv 1 \pmod{N}.$$

In other words

$$p^2q^4r^4 \equiv 1 \pmod{N}.$$

Hence if $X = pq^2r^2$ and $Y = 1$ then we obtain

$$X^2 = Y^2 \pmod{N}$$

as required and computing

$$\gcd(X - Y, N)$$

will give us a 50 percent chance of factoring N.

Whilst it was easy to see by inspection in the previous example how to combine the relations to obtain a square, in a real-life example our factorbase could consist of hundreds of thousands of primes and we would have hundreds of thousands of relations. We basically need a technique of automating this process of finding out how to combine relations into squares. This is where linear algebra can come to our aid.

We explain how to automate the process using linear algebra by referring to our previous simple example. Recall that our relations were equivalent to

$$\begin{aligned} p^{-1}qr^{-1} &= 1 \pmod N, \\ q^2r^3 &= 1 \pmod N, \\ p^2q^2r &= 1 \pmod N. \end{aligned}$$

To find which equations to multiply together to obtain a square we take a matrix A with $\#F$ columns and number of rows equal to the number of equations. Each equation is coded into the matrix as a row, modulo two, which in our example becomes

$$A = \begin{pmatrix} -1 & 1 & 1 \\ 0 & 2 & 3 \\ 2 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \pmod 2.$$

We now try and find a binary vector z such that

$$zA = 0 \pmod 2.$$

In our example we can take

$$z = (0, 1, 1)$$

since

$$\begin{pmatrix} 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \pmod 2.$$

This solution vector $z = (0, 1, 1)$ tells us that multiplying the last two equations together will produce a square modulo N.

Finding the vector z is done using a variant of Gaussian Elimination. Hence in general this means that we require more equations (i.e. relations) than elements in the factorbase. This relation combining stage of factoring algorithms is usually the hardest part since the matrices involved tend to be rather large. For example using the Number Field Sieve to factor a 100 decimal digit number may require a matrix of dimension over 100 000. This results in huge memory problems and requires the writing of specialist matrix code and often the use of specialized super computers.

The matrix will, for cryptographically interesting numbers, have around 500 000 rows and as many columns. As this is nothing but a matrix modulo 2 each entry could be represented by a single bit. If we used a dense matrix representation then the matrix alone would occupy around 29 gigabytes of storage. Luckily the matrix is very, very sparse and so the storage will not be so large.

As we said above we can compute the vector z such that $zA = 0$ using a variant of Gaussian Elimination over $\mathbb{Z}/2\mathbb{Z}$. But standard Gaussian Elimination would start with a sparse matrix and end up with an upper triangular dense matrix, so we would be back with the huge memory problem again. To overcome this problem very advanced matrix algorithms are deployed which try not to alter the matrix at all. We do not discuss these here but refer the interested reader to the book of Lenstra and Lenstra mentioned in the Further Reading section of this chapter.

The only thing we have not sketched is how to find the relations, a topic which we shall discuss in the next section.

4. Number Field Sieve

The Number Field Sieve is the fastest known factoring algorithm. The basic idea is to factor a number N by finding two integers x and y such that

$$x^2 \equiv y^2 \pmod{N};$$

we then expect (hope) that $\gcd(x-y, N)$ will give us a non-trivial factor of N.

To explain the basic method we shall start with the linear sieve and then show how this is generalized to the number field sieve. The linear sieve is not a very good algorithm but it does show the rough method.

4.1. The Linear Sieve. We let F denote a set of 'small' prime numbers which form the factorbase:

$$F = \{p : p \leq B\}.$$

A number which factorizes with all its factors in F is therefore B-smooth. The idea of the linear sieve is to find two integers a and λ such that

$$b = a + N\lambda$$

is B-smooth. If in addition we only select values of a which are 'small', then we would expect that a will also be B-smooth and we could write

$$a = \prod_{p \in F} p^{a_p}$$

and

$$b = a + N\lambda = \prod_{p \in F} p^{b_p}.$$

We would then have a relation in $\mathbb{Z}/N\mathbb{Z}$

$$\prod_{p \in F} p^{a_p} \equiv \prod_{p \in F} p^{b_p} \pmod{N}.$$

So the main question is how do we find such values of a and λ?

(1) Fix a value of λ to consider.
(2) Initialize an array of length $A+1$ indexed by 0 to A with zeros.
(3) For each prime $p \in F$ add $\log_2 p$ to every array location whose position is congruent to $-\lambda N \pmod{p}$.
(4) Choose the a to be the position of those elements which exceed some threshold bound.

The reasoning behind this method is that a position of the array which has an entry exceeding some bound, will when added to λN have a good chance of being B-smooth as it likely to be divisible by many primes in F.

For example suppose we take $N = 1159$, $F = \{2, 3, 5, 7, 11\}$ and $\lambda = -2$. So we wish to find a smooth value of

$$a - 2N.$$

We initialize the sieving array as follows:

0	1	2	3	4	5	6	7	8	9
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

We now take the first prime in F, namely $p = 2$, and we compute

$$-\lambda N \pmod{p} = 0.$$

So we add $\log_2(2) = 1$ to every array location with index equal to 0 modulo 2. This results in our sieve array becoming:

0	1	2	3	4	5	6	7	8	9
1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0

We now take the next prime in F, namely $p = 3$, and compute

$$-\lambda N \pmod{p} = 2.$$

So we add $\log_2(3) = 1.6$ onto every array location with index equal to 2 modulo 3. Our sieve array then becomes:

0	1	2	3	4	5	6	7	8	9
1.0	0.0	2.6	0.0	1.0	1.6	1.0	0.0	2.6	0.0

Continuing in this way with $p = 5, 7$ and 11, eventually the sieve array becomes:

0	1	2	3	4	5	6	7	8	9
1.0	2.8	2.6	2.3	1.0	1.6	1.0	0.0	11.2	0.0

Hence, the value $a = 8$ looks like it should correspond to a smooth value, and indeed it does, since we find

$$a - \lambda N = 8 - 2 \cdot 1159 = -2310 = -2 \cdot 3 \cdot 5 \cdot 7 \cdot 11.$$

So using the linear sieve we obtain a large collection of numbers a and b such that

$$a_i = \prod_{p_j \in F} p_j^{a_{i,j}} \equiv \prod_{p_j \in F} p_j^{b_{i,j}} = b_i \pmod{N}.$$

We assume that we have at least $|B| + 1$ such relations with which we then form a matrix with the rows

$$(a_{i,1}, \ldots, a_{i,t}, b_{i,1}, \ldots, b_{i,t}) \pmod{2}.$$

We then find elements of the kernel of this matrix modulo 2. This will tell us how to multiply the a_i and the b_i together to obtain elements x^2 and y^2 such that $x, y \in \mathbb{Z}$ are easily calculated and

$$x^2 \equiv y^2 \pmod{N}.$$

We can then try and factor N, but if these values of x and y do not provide a factor we just find a new element in the kernel of the matrix and continue.

The basic linear sieve gives a very small yield of relations. There is a variant called the *large prime variation* which relaxes the sieving condition to allow through pairs a and b which are almost B-smooth, bar say a single 'large' prime in a and a single 'large' prime in b. These large primes then have to be combined in some way so that the linear algebra step can proceed as above. This is done by constructing a graph and using an algorithm which computes a basis for the set of cycles in the graph. The basic idea for the large prime variation originally arose in the context of the Quadratic Sieve algorithm, but it can be applied to any of the sieving algorithms used in factoring.

It is clear that the sieving could be carried out in parallel, hence the sieving can be parcelled out to lots of slave computers around the world. The slaves then communicate any relations they find to the central master computer which performs the linear algebra step. In such a way the Internet can be turned into a large parallel computer dedicated to factoring numbers. The final linear algebra step we have already remarked needs to often be performed on specialized equipment with large amounts of disk space and RAM, so this final computation cannot be distributed over the Internet.

4.2. The Number Field Sieve. The linear sieve is simply not good enough to factor large numbers. Indeed the linear sieve was never proposed as a real factoring algorithm, its operation is however instructive for other algorithms of this type. The number field sieve uses the arithmetic of algebraic number fields to construct the desired relations between the elements of the factorbase. All that changes is the way the relations are found. The linear algebra step, the large prime variations and the slave/master approach all go over to NFS virtually unchanged. We now explain the NFS, but in a much simpler form than is actually used in real-life so as to aid the exposition. Those readers who do not know any algebraic number theory may wish to skip this section.

First we construct two monic, irreducible polynomials with integer coefficients f_1 and f_2, of degree d_1 and d_2 respectively, such that there exists an $m \in \mathbb{Z}$ such that

$$f_1(m) \equiv f_2(m) \equiv 0 \pmod{N}.$$

The number field sieve will make use of arithmetic in the number fields K_1 and K_2 given by

$$K_1 = \mathbb{Q}(\theta_1) \text{ and } K_2 = \mathbb{Q}(\theta_2),$$

where θ_1 and θ_2 are defined by $f_1(\theta_1) = f_2(\theta_2) = 0$. We then have two homomorphisms ϕ_1 and ϕ_2 given by

$$\phi_i : \begin{cases} \mathbb{Z}[\theta_i] \longrightarrow \mathbb{Z}/N\mathbb{Z} \\ \theta_i \longmapsto m. \end{cases}$$

We aim to use a sieve, just as in the linear sieve, to find a set

$$S \subset \{(a,b) \in \mathbb{Z}^2 : \gcd(a,b) = 1\}$$

such that

$$\prod_S (a - b\theta_1) = \beta^2$$

and

$$\prod_S (a - b\theta_2) = \gamma^2,$$

where $\beta \in K_1$ and $\gamma \in K_2$. If we found two such values of β and γ then we would have

$$\phi_1(\beta)^2 \equiv \phi_2(\gamma)^2 \pmod{N}$$

and hopefully

$$\gcd(N, \phi_1(\beta) - \phi_2(\gamma))$$

would be a factor of N.

This leads to three obvious problems

- How do we find the set S?
- Given $\beta^2 \in \mathbb{Q}[\theta_1]$, how do we compute β?
- How do we find the polynomials f_1 and f_2 in the first place?

4.2.1. *How do we find the set S?:* Similar to the linear sieve we can find such a set S using linear algebra provided we can find lots of a and b such that

$$a - b\theta_1 \text{ and } a - b\theta_2$$

are both 'smooth'. But what does it mean for these two objects to be smooth? It is here that the theory of algebraic number fields comes in, by generalizing our earlier definition of smooth integers to algebraic integers we obtain the definition:

DEFINITION 12.5. *An algebraic integer is 'smooth' if and only if the ideal it generates is only divisible by 'small' prime ideals.*

Define $F_i(X,Y) = Y^{d_i} f_i(X/Y)$ then

$$N_{\mathbb{Q}(\theta_i)/\mathbb{Q}}(a - b\theta_i) = F_i(a,b).$$

We define two factorbases, one for each of the polynomials

$$\mathcal{F}_i = \{(p,r) : p \text{ a prime}, r \in \mathbb{Z} \text{ such that } f_i(r) \equiv 0 \pmod{p}\}.$$

Each element of $\mathcal{F}_i$ corresponds to a degree one prime ideal of $\mathbb{Z}[\theta_i]$, which is a sub-order of the ring of integers of $\mathcal{O}_{\mathbb{Q}(\theta_i)}$, given by

$$\langle p, \theta_i - r\rangle = p\mathbb{Z}[\theta_i] + (\theta_i - r)\mathbb{Z}[\theta_i].$$

Given values of a and b we can easily determine whether the ideal $\langle a - \theta_i b\rangle$ 'factorizes' over our factorbase. Note factorizes is in quotes as unique factorization of ideals may not hold in $\mathbb{Z}[\theta_i]$, whilst it will hold in $\mathcal{O}_{\mathbb{Q}(\theta_i)}$. It will turn out that this is not really a problem. To see why this is not a problem you should consult the book by Lenstra and Lenstra.

If $\mathbb{Z}[\theta_i] = \mathcal{O}_{\mathbb{Q}(\theta_i)}$ then the following method does indeed give the unique prime ideal factorization of $\langle a - \theta_i b\rangle$.

- Write
$$F_i(a,b) = \prod_{(p_j,r)\in\mathcal{F}_i} p_j^{s_j^{(i)}}.$$
- We have $(a : b) = (r : 1) \pmod{p}$, as an element in the projective space of dimension one over $\mathbb{F}_p$, if the ideal corresponding to (p,r) is included in a non-trivial way in the ideal factorization of $a - \theta_i b$.
- We have
$$\langle a - \theta_i b\rangle = \prod_{(p_j,r)\in\mathcal{F}_i} \langle p_j, \theta_i - r\rangle^{s_j^{(i)}}.$$

This leads to the following algorithm to sieve for values of a and b, such that $\langle a - \theta_i b\rangle$ is an ideal which factorizes over the factorbase. Just as with the linear sieve, the use of sieving allows us to avoid lots of expensive trial divisions when trying to determine smooth ideals. We end up only performing factorizations where we already know we have a good chance of being successful.

- Fix a.
- Initialize the sieve array for $-B \le b \le B$ by
$$S[b] = \log_2(F_1(a,b) \cdot F_2(a,b)).$$
- For every $(p,r) \in \mathcal{F}_i$ subtract $\log_2 p$ from every array element such that
$$a - rb \equiv 0 \pmod{p}.$$
- The bs we want are the ones such that $S[b]$ lies below some tolerance level.

If the tolerance level is set in a sensible way then we have a good chance that both $F_1(a,b)$ and $F_2(a,b)$ factor over the prime ideals in the factorbase, with the possibility of some large prime ideals creeping in. We keep these factorizations as a relation, just as we did with the linear sieve.

Then, after some linear algebra, we can find a subset S of all the pairs (a,b) we have found such that

$$\prod_{(a,b)\in S} \langle a - b\theta_i\rangle = \text{square of an } \textbf{ideal} \text{ in } \mathbb{Z}[\theta_i].$$

This is not however good enough, recall we want the product $\prod a - b\theta_i$ to be the square of an **element** of $\mathbb{Z}[\theta_i]$. To overcome this problem we need to add information from the 'infinite' places. This is done by adding in some quadratic characters, an idea introduced by Adleman. Let q be

a rational prime (in neither $\mathcal{F}_1$ nor $\mathcal{F}_2$) such that there is an s_q with $f_i(s_q) \equiv 0 \pmod{q}$ and $f_i'(s_q) \not\equiv 0 \pmod{q}$ for either $i = 1$ or $i = 2$. Then our extra condition is that we require

$$\prod_{(a,b)\in S} \left(\frac{a - bs_q}{q}\right) = 1,$$

where $(\frac{\cdot}{q})$ denotes the Legendre symbol. As the Legendre symbol is multiplicative this gives us an extra condition to put into our matrix. We need to add this condition for a number of primes q, hence we choose a set of such primes q and put the associated characters into our matrix as an extra column of 0s or 1s corresponding to:

$$\text{if } \left(\frac{a - bs_q}{q}\right) = \begin{cases} 1 & \text{then enter } 0, \\ -1 & \text{then enter } 1. \end{cases}$$

So after finding enough relations we hope to be able to find a subset S such that hopefully

$$\prod_S (a - b\theta_1) = \beta^2 \text{ and } \prod_S (a - b\theta_2) = \gamma^2.$$

4.2.2. *How do we take the square roots?:* We then need to be able to take the square root of β^2 to recover β, and similarly for γ^2. Each β^2 is given in the form

$$\beta^2 = \sum_{j=0}^{d_1-1} a_j \theta_1^j$$

where the a_j are huge integers. We want to be able to determine the solutions $b_j \in \mathbb{Z}$ to the equation

$$\left(\sum_{j=0}^{d_1-1} b_j \theta_1^j\right)^2 = \sum_{j=0}^{d_1-1} a_j \theta_1^j.$$

One way this is overcome, due to Couveignes, is by computing such a square root modulo a large number of very, very large primes p. We then perform Hensel lifting and Chinese remaindering to hopefully recover our square root. This is the easiest method to understand although more advanced methods are available, such as that by Nguyen.

4.2.3. *Choosing the initial polynomials:* This is the part of the method that is a black art at the moment. We require only the following conditions to be met

$$f_1(m) \equiv f_2(m) \equiv 0 \pmod{N}.$$

However there are good heuristic reasons why it also might be desirable to construct polynomials with additional properties such as

- The polynomials have small coefficients.
- f_1 and f_2 have 'many' real roots. Note, a random polynomial probably would have no real roots on average.
- f_1 and f_2 have 'many' roots modulo lots of small prime numbers.
- The Galois groups of f_1 and f_2 are 'small'

It is often worth spending a few weeks trying to find a good couple of polynomials before one starts to attempt the factorization algorithm proper. There are a number of search strategies used for finding these polynomials. Once a few candidates are found some experimental sieving is performed to see which appear to be the most successful, in that they yield the most relations. Then once a decision has been made one can launch the sieving stage 'for real'.

4.3. Example. I am grateful to Richard Pinch for allowing me to include the following example. It is taken from his lecture notes from a course at Cambridge in the mid-1990s. Suppose we wish to factor the number $N = 290^2 + 1 = 84\,101$. We take $f_1(x) = x^2 + 1$ and $f_2(x) = x - 290$ with $m = 290$. Then

$$f_1(m) \equiv f_2(m) \equiv 0 \pmod{N}.$$

On one side we have the order $\mathbb{Z}[i]$ which is the ring of integers of $\mathbb{Q}(i)$ and on the other side we have the order $\mathbb{Z}$.

We obtain the following factorizations:

x	y	$N(x - iy)$	Factors	$x - my$	Factors
-38	-1	1445	$(5)(17^2)$	252	$(2^2)(3^2)(7)$
-22	-19	845	$(5)(13^2)$	(5488)	$(2^4)(7^3)$

We then obtain the two factorizations, which are real factorizations of elements, as $\mathbb{Z}[i]$ is a unique factorization domain,

$$-38 + i = -(2 + i)(4 - i)^2,\ -22 + 19i = -(2 + i)(3 - 2i)^2.$$

Hence, after a trivial bit of linear algebra, we obtain the following 'squares'

$$(-38 + i)(-22 + 19i) = (2 + i)^2(3 - 2i)^2(4 - i)^2 = (31 - 12i)^2$$

and

$$(-38 + m)(-22 + 19 \times m) = (2^6)(3^2)(7^4) = 1176^2.$$

We then apply the map ϕ_1 to $31 - 12i$ to obtain

$$\phi_1(31 - 12i) = 31 - 12 \times m = -3449.$$

But then we have

$$\begin{aligned}(-3449)^2 &= \phi_1(31 - 12i)^2 \\ &= \phi_1((31 - 12i)^2) \\ &= \phi_1((-38 + i)(-22 + 19i)) \\ &= \phi_1(-38 + i)\phi_1(-22 + 19i) \\ &\equiv (-38 + m)(-22 + 19 \times m) \pmod{N} \\ &= 1176^2.\end{aligned}$$

So we compute

$$\gcd(N, -3449 + 1176) = 2273$$

and

$$\gcd(N, -3449 - 1176) = 37.$$

Hence 37 and 2273 are factors of $N = 84\,101$.

Chapter Summary

- Prime numbers are very common and the probability that a random n-bit number is prime is around $1/n$.
- Numbers can be tested for primality using a probable prime test such as the Fermat or Miller–Rabin algorithms. The Fermat test has a problem in that certain composite numbers will always pass the Fermat test no matter how one chooses the possible witnesses.

- If one really needs to be certain that a number is prime then there are primality proving algorithms which run in polynomial time.
- Factoring algorithms are often based on the problem of finding the difference of two squares.
- Modern factoring algorithms run in two stages: In the first stage one collects a lot of relations on a factorbase by using a process called sieving, which can be done using thousands of computers on the Internet. In the second stage these relations are processed using linear algebra on a big central server. The final factorization is obtained by finding a difference of two squares.

Further Reading

The definitive reference work on computational number theory which deals with many algorithms for factoring and primality proving is the book by Cohen. The book by Bach and Shallit also provides a good reference for primality testing. The main book explaining the Number Field Sieve is the book by Lenstra and Lenstra.

E. Bach and J. Shallit. *Algorithmic Number Theory. Volume 1: Efficient Algorithms.* MIT Press, 1996.

H. Cohen. *A Course in Computational Algebraic Number Theory.* Springer-Verlag, 1993.

A. Lenstra and H. Lenstra. *The Development of the Number Field Sieve.* Springer-Verlag, 1993.

CHAPTER 13

Discrete Logarithms

Chapter Goals

- To examine algorithms for solving the discrete logarithm problem.
- To introduce the Pohlig–Hellman algorithm.
- To introduce the Baby-Step/Giant-Step algorithm.
- To explain the methods of Pollard.
- To show how discrete logarithms can be solved in finite fields using algorithms like those used for factoring.
- To describe the known results on the elliptic curve discrete logarithm problem.

1. Introduction

In this chapter we survey the methods known for solving the discrete logarithm problem,

$$h = g^x$$

in various groups G. These algorithms fall into one of two categories, either the algorithms are generic and apply to any finite abelian group or the algorithms are specific to the special group under consideration.

We start with general purpose algorithms and then move onto special purpose algorithms later in the chapter.

2. Pohlig–Hellman

The first observation to make is that the discrete logarithm problem in a group G is only as hard as the discrete logarithm problem in the largest subgroup of prime order in G. This observation is due to Pohlig and Hellman, and it applies in an arbitrary finite abelian group.

To explain the Pohlig–Hellman algorithm, suppose we have a finite cyclic abelian group $G = \langle g \rangle$ whose order is given by

$$N = \#G = \prod_{i=1}^{t} p_i^{e_i}.$$

Now suppose we are given $h \in \langle g \rangle$, so there exists an integer x such that

$$h = g^x.$$

Our aim is to find x by first finding it modulo $p_i^{e_i}$ and then using the Chinese Remainder Theorem to recover it modulo N.

From basic group theory we know that there is a group isomorphism

$$\phi : G \longrightarrow C_{p_1^{e_1}} \times \cdots \times C_{p_t^{e_t}},$$

where C_{p^e} is a cyclic group of prime power order p^e. The projection of ϕ to the component C_{p^e} is given by

$$\phi_p : \begin{cases} G \longrightarrow C_{p^e} \\ f \longmapsto f^{N/p^e}. \end{cases}$$

Now the map ϕ_p is a group homomorphism so if we have $h = g^x$ in G then we will have $\phi_p(h) = \phi_p(g)^x$ in C_{p^e}. But the discrete logarithm in C_{p^e} is only determined modulo p^e. So if we could solve the discrete logarithm problem in C_{p^e}, then we would determine x modulo p^e. Doing this for all primes p dividing N would allow us to solve for x using the Chinese Remainder Theorem.

In summary suppose we had some oracle $O(g, h, p, e)$ which for $g, h \in C_{p^e}$ will output the discrete logarithm of h with respect to g. We can then solve for x using Algorithm 13.1

Algorithm 13.1: Algorithm to find DLP in group of order N, given an oracle for DLP for prime power divisors of N

```
S = {}
forall primes p dividing N do
    Compute the largest e such that T = p^e divides N
    g1 = g^{N/T}
    h1 = h^{N/T}
    z = O(g1, h1, p, e)
    S = S + {(z, T)}
end
x =CRT(S)
```

The only problem is that we have not shown how to solve the discrete logarithm problem in C_{p^e}. We shall now show how this is done, by reducing to solving e discrete logarithm problems in the group C_p. Suppose $g, h \in C_{p^e}$ and that there is an x such that

$$h = g^x.$$

Clearly x is only defined modulo p^e and we can write

$$x = x_0 + x_1 p + \cdots + x_{e-1} p^{e-1}.$$

We find $x_0, x_1, \ldots$ in turn, using the following inductive procedure. Suppose we know x', the value of x modulo p^t, i.e.

$$x' = x_0 + \cdots + x_{t-1} p^{t-1}.$$

We now wish to determine x_t and so compute x modulo p^{t+1}. We write

$$x = x' + p^t x'',$$

so we have that

$$h = g^{x'} (g^{p^t})^{x''}.$$

Hence, if we set

$$h' = h g^{-x'} \text{ and } g' = g^{p^t},$$

then

$$h' = g'^{x''}.$$

Now g' is an element of order p^{e-t} so to obtain an element of order p, and hence a discrete logarithm problem in C_p we need to raise the above equation to the power $s = p^{e-t-1}$. So setting

$$h'' = h'^s \text{ and } g'' = g'^s$$

we obtain the discrete logarithm problem in C_p given by

$$h'' = g''^{x_t}.$$

So assuming we can solve discrete logarithms in C_p we can find x_t and so find x.

We now illustrate this approach, assuming we can solve discrete logarithms in cyclic groups of prime order. We leave to the next two sections techniques to find discrete logarithms in cyclic groups of prime order, for now we assume that this is possible. As an example of the Pohlig–Hellman algorithm, consider the multiplicative group of the finite field $\mathbb{F}_{397}$, this has order

$$396 = 2^2 \cdot 3^2 \cdot 11$$

and a generator of $\mathbb{F}_{397}^*$ is given by

$$g = 5.$$

We wish to solve the discrete logarithm problem given by

$$h = 208 = 5^x \pmod{397}.$$

We first reduce to the three subgroups of prime power order, by raising the above equation to the power $396/p^e$. Hence, we obtain the three discrete logarithm problems

$$\begin{aligned} 334 &= h^{396/4} = g^{396/4 x_4} = 334^{x_4} \pmod{397}, \\ 286 &= h^{396/9} = g^{396/9 x_9} = 79^{x_9} \pmod{397}, \\ 273 &= h^{396/11} = g^{396/11 x_{11}} = 290^{x_{11}} \pmod{397}. \end{aligned}$$

The value of x_4 is the value of x modulo 4, the value of x_9 is the value of x modulo 9 whilst the value of x_{11} is the value of x modulo 11. Clearly if we can determine these three values then we can determine x modulo 396.

2.1. Determining x_4. By inspection we see that $x_4 = 1$, but let us labour the point and show how the above algorithm will determine this for us. We write

$$x_4 = x_{4,0} + 2 \cdot x_{4,1},$$

where $x_{4,0}, x_{4,1} \in \{0, 1\}$. Recall that we wish to solve

$$h' = 334 = 334^{x_4} = g'^{x_4}.$$

We set $h'' = h'^2$ and $g'' = g'^2$ and solve the discrete logarithm problem

$$h'' = g''^{x_{4,0}}$$

in the cyclic group of order 2. We find, using our oracle for the discrete logarithm problem in cyclic groups, that $x_{4,0} = 1$. So we now have

$$\frac{h'}{g'} = g''^{x_{4,1}} \pmod{397}.$$

Hence we have

$$1 = 396^{x_{4,1}},$$

which is another discrete logarithm in the cyclic group of order two. We find $x_{4,1} = 0$ and so conclude, as expected, that

$$x_4 = x_{4,0} + 2 \cdot x_{4,1} = 1 + 2 \cdot 0 = 1.$$

2.2. Determining x_9. We write

$$x_9 = x_{9,0} + 3 \cdot x_{9,1},$$

where $x_{9,0}, x_{9,1} \in \{0, 1, 2\}$. Recall that we wish to solve

$$h' = 286 = 79^{x_9} = g'^{x_9}.$$

We set $h'' = h'^3$ and $g'' = g'^3$ and solve the discrete logarithm problem

$$h'' = 34 = g''^{x_{9,0}} = 362^{x_{9,0}}$$

in the cyclic group of order 3. We find, using our oracle for the discrete logarithm problem in cyclic groups, that $x_{9,0} = 2$. So we now have

$$\frac{h'}{g'^2} = g''^{x_{9,1}} \pmod{397}.$$

Hence we have

$$1 = 362^{x_{9,1}},$$

which is another discrete logarithm in the cyclic group of order two. We find $x_{9,1} = 0$ and so conclude that

$$x_9 = x_{9,0} + 3 \cdot x_{9,1} = 2 + 3 \cdot 0 = 2.$$

2.3. Determining x_{11}. We are already in a cyclic group of prime order, so applying our oracle to the discrete logarithm problem

$$273 = 290^{x_{11}} \pmod{397},$$

we find that $x_{11} = 6$.

So we have determined that if

$$208 = 5^x \pmod{397},$$

then x is given by

$$\begin{aligned} x &= 1 \pmod 4, \\ x &= 2 \pmod 9, \\ x &= 6 \pmod{11}. \end{aligned}$$

If we apply the Chinese Remainder Theorem to this set of three simultaneous equations then we obtain that the solution to our discrete logarithm problem is given by

$$x = 281.$$

3. Baby-Step/Giant-Step Method

In our above discussion of the Pohlig–Hellman algorithm we assumed we had an oracle to solve the discrete logarithm problem in cyclic groups of prime order. We shall now describe a general method of solving such problems due to Shanks called the Baby-Step/Giant-Step method. Once again this is a generic method which applies to any cyclic finite abelian group.

Since the intermediate steps in the Pohlig–Hellman algorithm are quite simple, the difficulty of solving a general discrete logarithm problem will be dominated by the time required to solve the discrete logarithm problem in the cyclic subgroups of prime order. Hence, for generic groups the complexity of the Baby-Step/Giant-Step method will dominate the overall complexity of any algorithm. Indeed one can show that the following method is the best possible method, time-wise, for solving the discrete logarithm problem in an arbitrary group. Of course in any actual group there may be a special purpose algorithm which works faster, but in general the following is provably the best one can do.

Again we fix notation as follows: We have a public cyclic group $G = \langle g \rangle$, which we can now assume to have prime order p. We are also given an $h \in G$ and are asked to find the value of x modulo p such that

$$h = g^x.$$

We assume there is some fixed encoding of the elements of G, so in particular it is easy to store, sort and search a list of elements of G.

The idea behind the Baby-Step/Giant-Step method is a standard divide and conquer approach found in many areas of computer science. We first write

$$x = x_0 + x_1 \lceil \sqrt{p} \rceil.$$

Now, since $x \leq p$, we have that $0 \leq x_0, x_1 < \lceil \sqrt{p} \rceil$.

We first compute the Baby-Steps

$$g_i = g^i \text{ for } 0 \leq i < \lceil \sqrt{p} \rceil.$$

The pairs

$$(g_i, i)$$

are stored in a table so that one can easily search for items indexed by the first entry in the pair. This can be accomplished by sorting the table on the first entry or more efficiently by the use of hash-tables. To compute and store the Baby-Steps clearly requires

$$O(\lceil \sqrt{p} \rceil)$$

time, and a similar amount of storage.

We now compute the Giant-Steps

$$h_j = hg^{-j\lceil \sqrt{p} \rceil} \text{ for } 0 \leq j < \lceil \sqrt{p} \rceil.$$

We then try to find a match in the table of Baby-Steps, i.e. we try to find a value g_i such that $g_i = h_j$. If such a match occurs we have

$$x_0 = i \text{ and } x_1 = j$$

since, if $g_i = h_j$,

$$g^i = hg^{-j\lceil \sqrt{p} \rceil},$$

i.e.

$$g^{i+j\lceil \sqrt{p} \rceil} = h.$$

Notice that the time to compute the Giant-Steps is at most

$$O(\lceil \sqrt{p} \rceil).$$

Hence, the overall time and space complexity of the Baby-Step/Giant-Step method is

$$O(\sqrt{p}).$$

This means, combining with the Pohlig–Hellman algorithm, that if we wish a discrete logarithm problem in a group G to be as difficult as a work effort of 2^{80} operations, then we need the group G to have a prime order subgroup of order larger than 2^{160}.

As an example we take the subgroup of order 101 in the multiplicative group of the finite field $\mathbb{F}_{607}$, generated by $g = 64$. Suppose we are given the discrete logarithm problem

$$h = 182 = 64^x \pmod{607}.$$

We first compute the Baby-Steps

$$g_i = 64^i \pmod{607} \text{ for } 0 \leq i < \lceil \sqrt{101} \rceil = 11.$$

We compute

i	$64^i \pmod{607}$	i	$64^i \pmod{607}$
0	1	6	330
1	64	7	482
2	454	8	498
3	527	9	308
4	343	10	288
5	100		

Now we compute the Giant-Steps,

$$h_j = 182 \cdot 64^{-11j} \pmod{607} \text{ for } 0 \leq j < 11,$$

and check when we obtain a Giant-Step which occurs in our table of Baby-Steps:

j	$182 \cdot 64^{-11j} \pmod{607}$	j	$182 \cdot 64^{-11j} \pmod{607}$
0	182	6	60
1	143	7	394
2	69	8	483
3	271	9	76
4	343	10	580
5	573		

So we obtain a match when $i = 4$ and $j = 4$, which means that

$$x = 4 + 11 \cdot 4 = 48,$$

which we can verify to be the correct answer to the earlier discrete logarithm problem by computing

$$64^{48} \pmod{607} = 182.$$

4. Pollard Type Methods

The trouble with the Baby-Step/Giant-Step method was that although its run time was bounded by $O(\sqrt{p})$ it also required $O(\sqrt{p})$ space. This space requirement is more of a hindrance in practice than the time requirement. Hence, one could ask whether one could trade the large space requirement for a smaller space requirement, but still obtain a time complexity of $O(\sqrt{p})$? Well we can, but we will now obtain only an expected running time rather than an absolute bound on the running time. There are a number of algorithms which achieve this reduced space requirement all of which are due to ideas of Pollard.

4.1. Pollard's Rho Algorithm. Suppose $f : S \to S$ is a random mapping between a set S and itself, where the size of S is n. Now pick a random value $x_0 \in S$ and compute

$$x_{i+1} = f(x_i) \text{ for } i \geq 0.$$

The values $x_0, x_1, x_2, \ldots$ we consider as a deterministic random walk. By this last statement we mean that each step $x_{i+1} = f(x_i)$ of the walk is a deterministic function of the current position x_i, but we are assuming that the sequence $x_0, x_1, x_2, \ldots$ *behaves* as a random sequence would. Another name for a deterministic random walk is a pseudo-random walk.

Since S is finite we must eventually obtain

$$x_i = x_j$$

and so

$$x_{i+1} = f(x_i) = f(x_j) = x_{j+1}.$$

Hence, the sequence $x_0, x_1, x_2, \ldots$, will eventually become cyclic. If we 'draw' such a sequence then it looks like the Greek letter rho, i.e.

$$\rho.$$

In other words there is a cyclic part and an initial tail. One can show that for a random mapping the tail has expected length (i.e. the number of elements in the tail)

$$\sqrt{\pi n/8},$$

whilst the cycle has expected length (i.e. the number of elements in the cycle)

$$\sqrt{\pi n/8}.$$

The goal of many of Pollard's algorithms is to find a collision in a random mapping like the one above. A collision is finding two values x_i and x_j with $i \neq j$ such that

$$x_i = x_j.$$

From the birthday paradox we obtain a collision after an expected number of

$$\sqrt{\pi n/2}$$

iterations of the map f. Hence, finding a collision using the birthday paradox in a naive way would require $O(\sqrt{n})$ time and $O(\sqrt{n})$ memory. But this is exactly the problem with the Baby-Step/Giant-Step method we were trying to avoid.

To find a collision, and make use of the rho shape of the random walk, we use Floyd's cycle finding algorithm: Given (x_1, x_2) we compute (x_2, x_4) and then (x_3, x_6) and so on, i.e. given the pair (x_i, x_{2i}) we compute

$$(x_{i+1}, x_{2i+2}) = (f(x_i), f(f(x_{2i}))).$$

We stop when we find

$$x_m = x_{2m}.$$

If the tail of the sequence $x_0, x_1, x_2, \ldots$ has length λ and the cycle has length μ then one can show that we obtain such a value of m when

$$m = \mu \left(1 + \lfloor \lambda/\mu \rfloor\right).$$

Since $\lambda < m \leq \lambda + \mu$ we see that

$$m = O(\sqrt{n}),$$

and this will be an accurate complexity estimate if the mapping f behaves like an average random function. Hence, we can detect a collision with virtually no storage.

This is all very well, but we have not shown how to relate this to the discrete logarithm problem: Let G denote a group of order n and let the discrete logarithm problem be given by

$$h = g^x.$$

We partition the group into three sets S_1, S_2, S_3, where we assume $1 \notin S_2$, and then define the following random walk on the group G,

$$x_{i+1} = f(x_i) = \begin{cases} h \cdot x_i & x_i \in S_1, \\ x_i^2 & x_i \in S_2, \\ g \cdot x_i & x_i \in S_3. \end{cases}$$

In practice we actually keep track of three pieces of information

$$(x_i, a_i, b_i)$$

where

$$a_{i+1} = \begin{cases} a_i & x_i \in S_1, \\ 2a_i \pmod{n} & x_i \in S_2, \\ a_i + 1 \pmod{n} & x_i \in S_3, \end{cases}$$

and

$$b_{i+1} = \begin{cases} b_i + 1 \pmod{n} & x_i \in S_1, \\ 2b_i \pmod{n} & x_i \in S_2, \\ b_i & x_i \in S_3. \end{cases}$$

If we start with the triple

$$(x_0, a_0, b_0) = (1, 0, 0)$$

then we have, for all i,

$$\log_g(x_i) = a_i + b_i \log_g(h) = a_i + b_i x.$$

Applying Floyd's cycle finding algorithm we obtain a collision, and so find a value of m such that

$$x_m = x_{2m}.$$

This leads us to deduce the following equality of discrete logarithms

$$\begin{aligned} a_m + b_m x &= a_m + b_m \log_g(h) \\ &= \log_g(x_m) \\ &= \log_g(x_{2m}) \\ &= a_{2m} + b_{2m} \log_g(h) \\ &= a_{2m} + b_{2m} x. \end{aligned}$$

Rearranging we see that

$$(b_m - b_{2m})x = a_{2m} - a_m,$$

and so, if $b_m \neq b_{2m}$, we obtain

$$x = \frac{a_{2m} - a_m}{b_m - b_{2m}} \pmod{n}.$$

The probability that we have $b_m = b_{2m}$ is small enough, for large n, to be ignored.

If we assume the sequence $x_0, x_1, x_2, \ldots$ is produced by a random mapping from the group G to itself, then the above algorithm will find the discrete logarithm in expected time

$$O(\sqrt{n}).$$

As an example consider the subgroup G of $\mathbb{F}_{607}^*$ of order $n = 101$ generated by the element $g = 64$ and the discrete logarithm problem

$$h = 122 = 64^x.$$

We define the sets S_1, S_2, S_3 as follows:

$$\begin{aligned} S_1 &= \{x \in \mathbb{F}_{607}^* : x \leq 201\}, \\ S_2 &= \{x \in \mathbb{F}_{607}^* : 202 \leq x \leq 403\}, \\ S_3 &= \{x \in \mathbb{F}_{607}^* : 404 \leq x \leq 606\}. \end{aligned}$$

Applying Pollard's Rho method we obtain the following data

i	x_i	a_i	b_i	x_{2i}	a_{2i}	b_{2i}
0	1	0	0	1	0	0
1	122	0	1	316	0	2
2	316	0	2	172	0	8
3	308	0	4	137	0	18
4	172	0	8	7	0	38
5	346	0	9	309	0	78
6	137	0	18	352	0	56
7	325	0	19	167	0	12
8	7	0	38	498	0	26
9	247	0	39	172	2	52
10	309	0	78	137	4	5
11	182	0	55	7	8	12
12	352	0	56	309	16	26
13	76	0	11	352	32	53
14	167	0	12	167	64	6

So we obtain a collision, using Floyd's cycle finding algorithm, when $m = 14$. We see that

$$g^0 h^{12} = g^{64} h^6$$

which implies

$$12x = 64 + 6x \pmod{101}.$$

In other words

$$x = \frac{64}{12 - 6} \pmod{101} = 78.$$

4.2. Pollard's Lambda Method. Pollard's Lambda method is like the Rho method in that one uses a deterministic random walk and a small amount of storage to solve the discrete logarithm problem. However, the Lambda method is particularly tuned to the situation where one knows that the discrete logarithm lies in a certain interval

$$x \in [a, \ldots, b].$$

In the Rho method we used one random walk, which turned into the shape of the Greek letter ρ, whilst in the Lambda method we use two walks which end up in the shape of the Greek letter lambda, i.e.

$$\lambda,$$

hence giving the method its name. Another name for this method is Pollard's Kangaroo method as it is originally described by the two walks being performed by kangaroos.

Let $w = b - a$ denote the length of the interval in which the discrete logarithm x is known to lie. We define a set

$$S = \{s_0, \ldots, s_{k-1}\}$$

of integers in non-decreasing order. The mean m of the set should be around $N = \sqrt{w}$. It is common to choose

$$s_i = 2^i \text{ for } 0 \leq i < k,$$

which implies that the mean of the set is

$$\frac{2^k}{k}$$

and so we choose

$$k \approx \frac{1}{2} \cdot \log_2(w).$$

We divide the group up into k sets S_i, for $i = 0, \ldots, k-1$ and define the following deterministic random walk:

$$x_{i+1} = x_i \cdot g^{s_j} \text{ if } x_i \in S_j.$$

We first compute the deterministic random walk, starting from $g_0 = g^b$, by setting

$$g_i = g_{i-1} \cdot g^{s_j}$$

for $i = 1, \ldots, N$. We also set $c_0 = b$ and $c_{i+1} = c_i + s_j \pmod{q}$. We store g_N and notice that we have computed the discrete logarithm of g_N with respect to g,

$$c_N = \log_g(g_N).$$

We now compute our second deterministic random walk starting from the unknown point in the interval x; we set $h_0 = h = g^x$ and compute

$$h_{i+1} = h_i \cdot g^{s'_j}.$$

We also set $d_0 = 0$ and $d_{i+1} = d_i + s'_j \pmod{q}$. Notice that we have

$$\log_g(h_i) = x + d_i.$$

If the path of the h_i meets that of the path of the g_i then the h_i will carry on the path of the g_i and we will be able to find a value M where h_M equals our stored point g_N. At this point we have

$$c_N = \log_g(g_N) = \log_g(h_M) = x + d_M,$$

and so the solution to our discrete logarithm problem is given by

$$x = c_N - d_M \pmod{q}.$$

If we do not get a collision then we can increase N and continue both walks in a similar manner until a collision does occur.

The expected running time of this method is $\sqrt{w}$ and again the storage can be seen to be constant. The Lambda method can be used when the discrete logarithm is only known to lie in the full interval $[0, \ldots, q-1]$. But in this situation, whilst the asymptotic complexity is the same as the Rho method, the Rho method is better due to the implied constants.

As an example we again consider the subgroup G of $\mathbb{F}_{607}^*$ of order $n = 101$ generated by the element $g = 64$, but now we look at the discrete logarithm problem

$$h = 524 = 64^x.$$

We are given that the discrete logarithm x lies in the interval $[60, \ldots, 80]$. As our set of multipliers s_i we take $s_i = 2^i$ for $i = 0, 1, 2, 3$. The subsets $S_0, \ldots, S_3$ of G we define by

$$S_i = \{g \in G : g \pmod 4 = i\}.$$

We first compute the deterministic random walk g_i and the discrete logarithms $c_i = \log_g(g_i)$, for $i = 0, \ldots, N = 4$.

i	g_i	c_i
0	151	80
1	537	88
2	391	90
3	478	98
4	64	1

Now we compute the second deterministic random walk

i	h_i	$d_i = \log_g(h_i) - x$
0	524	0
1	151	1
2	537	9
3	391	11
4	478	19
5	64	23

Hence, we obtain the collision

$$h_5 = g_4$$

and so

$$x = 1 - 23 \pmod{101} = 79.$$

Note that examining the above tables we see that we had earlier collisions between our two walks. However, we are unable to use these since we do not store g_0, g_1, g_2 or g_3. We have only stored the value of g_4.

4.3. Parallel Pollard's Rho. In real life when one uses random walk based techniques to solve discrete logarithm problems one uses a parallel version, so as to exploit the computing resources of a number of sites across the Internet. Suppose we are given the discrete logarithm problem

$$h = g^x$$

in a group G of prime order q. We first decide on an easily computable function

$$H : G \longrightarrow \{1, \ldots, k\},$$

where k is usually around 20. Then we define a set of multipliers m_i, these are produced by generating random integers $a_i, b_i \in [0, \ldots, q-1]$ and then setting

$$m_i = g^{a_i} h^{b_i}.$$

To start a deterministic random walk we pick random $s_0, t_0 \in [0, \ldots, q-1]$ and compute

$$g_0 = g^{s_0} h^{t_0},$$

the deterministic random walk is then defined on the triples (g_i, s_i, t_i) where

$$\begin{aligned} g_{i+1} &= g_i \cdot m_{H(g_i)}, \\ s_{i+1} &= s_i + a_{H(g_i)} \pmod{q}, \\ t_{i+1} &= t_i + b_{H(g_i)} \pmod{q}. \end{aligned}$$

Hence, for every g_i we record the values of s_i and t_i such that

$$g_i = g^{s_i} h^{t_i}.$$

Suppose we have m processors, each processor starts a different deterministic random walk from a different starting position using the same algorithm to determine the next element in the walk. When two processors, or even the same processor, meet an element of the group that has been seen before then we obtain an equation

$$g^{s_i} h^{t_i} = g^{s'_j} h^{t'_j}$$

from which we can solve for the discrete logarithm x. Hence, we expect that after $O(\sqrt{\pi q/2}/m)$ iterations of these parallel walks we will find a collision and so solve the discrete logarithm problem.

However, as described this means that each processor needs to return every element in its computed deterministic random walk to a central server which then stores all the computed elements. This is highly inefficient as the storage requirements will be very large, namely $O(\sqrt{\pi q/2})$. We can reduce the storage to any required value as follows:

We define a function d on the group

$$d : G \longrightarrow \{0, 1\}$$

such that $d(g) = 1$ around $1/2^t$ of the time. The function d is often defined by returning $d(g) = 1$ if a certain subset of t of the bits representing g are set to zero for example. The elements in G for which $d(g) = 1$ will be called distinguished.

It is only the distinguished group elements which are now transmitted back to the central server. This means that one expects the deterministic random walks to need to continue another 2^t steps before a collision is detected between two deterministic random walks. Hence, the computing time now becomes

$$O\left(\sqrt{\pi q/2}/m + 2^t\right),$$

whilst the storage becomes

$$O\left(\sqrt{\pi q/2}/2^t\right).$$

This allows the storage to be reduced to any manageable amount, at the expense of a little extra computation. We do not give an example, since the method only really becomes useful as q becomes larger (say $q > 2^{20}$), but we leave it to the reader to construct their own examples.

5. Sub-exponential Methods for Finite Fields

There is a close relationship between the sub-exponential methods for factoring and the sub-exponential methods for solving the discrete logarithm problem in finite fields. We shall only consider the case of prime fields $\mathbb{F}_p$ but similar considerations apply to finite fields of characteristic two. The sub-exponential algorithms for finite fields are often referred to as index-calculus algorithms, for a reason which will become apparent as we explain the method.

We assume we are given $g, h \in \mathbb{F}_p^*$ such that

$$h = g^x.$$

We choose a factor base F of elements, usually small prime numbers, and then using one of the sieving strategies used for factoring we obtain a large number of relations of the form

$$\prod_{p_i \in F} p_i^{e_i} = 1 \pmod{p}.$$

These relations translate into the following equations for discrete logarithms,

$$\sum_{p_i \in F} e_i \log_g(p_i) = 0 \pmod{p-1}.$$

Once enough equations like the one above have been found we can solve for the discrete logarithm of every element in the factor base, i.e. we can determine

$$x_i = \log_g(p_i).$$

The value of x_i is sometimes called the index of p_i with respect to g. This calculation is performed using linear algebra modulo $p-1$, which is hence more complicated than the linear algebra modulo 2 performed in factoring algorithms. However, similar tricks, such as those deployed in the linear algebra stage of factoring algorithms, can be deployed to keep the storage requirements down to manageable levels. This linear algebra calculation only needs to be done once for each generator g, the results can then be used for many values of h.

When one wishes to solve a particular discrete logarithm problem

$$h = g^x,$$

we use a sieving technique, or simple trial and error, to write

$$h = \prod_{p_i \in F} p_i^{h_i} \pmod{p},$$

i.e. we could compute

$$T = h \prod_{p_i \in F} p_i^{f_i} \pmod{p}$$

and see if it factors in the form

$$T = \prod_{p_i \in F} p_i^{g_i}.$$

If it does then we have

$$h = \prod_{p_i \in F} p_i^{g_i - f_i} \pmod{p}.$$

We can then compute the discrete logarithm x from

$$\begin{aligned} x = \log_g(h) &= \log_g \left(\prod_{p_i \in F} p_i^{h_i} \right) \\ &= \sum_{p_i \in F} h_i \log_g(p_i) \pmod{p-1} \\ &= \sum_{p_i \in F} h_i \cdot x_i \pmod{p-1}. \end{aligned}$$

This means that, once one discrete logarithm has been found, determining the next one is easier since we have already computed the values of the x_i.

The best of the methods to find the relations between the factorbase elements is the Number Field Sieve. This gives an overall running time of

$$O(L_p(1/3, c))$$

for some constant c. This is roughly the same complexity as the algorithms to factor large numbers, although the real practical problem is that the matrix algorithms now need to work modulo $p-1$ and not modulo 2 as they did in factoring algorithms.

The upshot of these sub-exponential methods is that the size of p for finite field discrete logarithm based systems needs to be of the same order of magnitude as an RSA modulus, i.e. $p \geq 2^{1024}$.

Even though p has to be very large we still need to guard against generic attacks, hence $p-1$ should have a prime factor q of order greater than 2^{160}. In fact for finite field based systems we usually work in the subgroup of $\mathbb{F}_p^*$ of order q.

6. Special Methods for Elliptic Curves

For elliptic curves there are no known sub-exponential methods for the discrete logarithm problem, except in certain special cases. This means that the only method to solve the discrete logarithm problem in this setting is the parallel version of Pollard's Rho method.

Suppose the elliptic curve E is defined over the finite field $\mathbb{F}_q$. We set

$$\#E(\mathbb{F}_q) = h \cdot r$$

where r is a prime number. By Hasse's Theorem 2.3 the value of $\#E(\mathbb{F}_q)$ is close to q so we typically choose a curve with r close to q, i.e. we choose a curve E so that $h = 1, 2$ or 4.

The best known general algorithm for the elliptic curve discrete logarithm problem is the parallel Pollard's Rho method, which has complexity $O(\sqrt{r})$, which is about $O(\sqrt{q})$. Hence, to achieve the same security as an 80-bit block cipher we need to take $q \approx 2^{160}$, which is a lot smaller than the

field size recommended for systems based on the discrete logarithm problems in a finite field. This results in the reduced bandwidth and computational times of elliptic curve systems.

However, there are a number of special cases which need to be avoided. We shall now give these special cases, but we shall not give the detailed reasons why they are to be avoided since the reasons are often quite mathematically involved. As usual we assume q is either a large prime or a power of 2.

- For any q we must choose curves for which there is no small number t such that r divides $q^t - 1$, where r is the large prime factor of $\#E(\mathbb{F}_q)$. This eliminates the supersingular curves and a few others. In this case there is a simple computable mapping from the elliptic curve discrete logarithm problem to the discrete logarithm problem in the finite field $\mathbb{F}_{q^t}$. Hence, in this case we obtain a sub-exponential method for solving the elliptic curve discrete logarithm problem.
- If $q = p$ is a large prime then we need to avoid the anomalous curves where $E(\mathbb{F}_p) = p$. In this case there is an algorithm which requires $O(\log p)$ elliptic curve operations.
- If $q = 2^n$ then we usually assume that n is prime to avoid the possibility of certain attacks based on the concept of 'Weil descent'.

One should treat these three special cases much like one treats the generation of large integers for the RSA algorithm. Due to the P $-$ 1 factoring method one often makes RSA moduli $N = p \cdot q$ such that p is a so-called safe prime of the form $2p_1 + 1$. Another special RSA based case is that we almost always use RSA with two prime factors, rather than three or four. This is because moduli with two prime factors appear to be the hardest to factor.

Chapter Summary

- Due to the Pohlig–Hellman algorithm a hard discrete logarithm problem should be set in a group whose order has a large prime factor.
- The generic algorithms such as the Baby-Step/Giant-Step algorithm mean that to achieve the same security as an 80-bit block cipher, the size of this large prime factor of the group order should be at least 160 bits.
- The Baby-Step/Giant-Step algorithm is a generic algorithm whose running time can be absolutely bounded by $\sqrt{q}$, where q is the size of the large prime factor of $\#G$. However, the storage requirements of the Baby-Step/Giant-Step algorithm are $O(\sqrt{q})$.
- There are a number of techniques, due to Pollard, based on deterministic random walks in a group. These are generic algorithms which require little storage but which solve the discrete logarithm problem in expected time $O(\sqrt{q})$.
- For finite fields a number of index calculus algorithms exist which run in sub-exponential time. These mean that one needs to take large finite fields $\mathbb{F}_{p^t}$ with $p^t \geq 2^{1024}$ if one wants to obtain a hard discrete logarithm problem.
- For elliptic curves there are no sub-exponential algorithms known, except in very special cases. Hence, the only practical general algorithm to solve the discrete logarithm problem on an elliptic curve is the parallel Pollard's Rho method.

Further Reading

There are a number of good surveys on the discrete logarithm problem. I would recommend the ones by McCurley and Odlyzko. These articles only touch on the elliptic curve discrete logarithm problem though. For a treatment of what is known on the latter problem you could consult the survey by Koblitz et al.

N. Koblitz, A. Menezes and S. Vanstone. *The state of elliptic curve cryptography.* Designs Codes and Cryptography, **19**, 173–193, 2000.

K. McCurley. *The discrete logarithm problem.* In *Cryptology and Computational Number Theory*, Proc. Symposia in Applied Maths, Volume 42, 1990.

A. Odlyzko. *Discrete logarithms: The past and the future.* Designs Codes and Cryptography, **19**, 129–145, 2000.

CHAPTER 14

Key Exchange and Signature Schemes

Chapter Goals

- To introduce Diffie–Hellman key exchange.
- To introduce the need for digital signatures.
- To explain the two most used signature algorithms, namely RSA and DSA.
- To explain the need for cryptographic hash functions within signature schemes.
- To describe some other signature algorithms and key exchange techniques which have interesting properties.

1. Diffie–Hellman Key Exchange

Recall that the main drawback with the use of fast bulk encryption based on block or stream ciphers was the problem of key distribution. We have already seen a number of techniques to solve this problem, either using protocols which are themselves based on symmetric key techniques, or using a public key algorithm to transport a session key to the intended recipient. These, however, both have problems associated with them. For example, the symmetric key protocols were hard to analyse and required the use of already deployed long-term keys between each user and a trusted central authority.

A system is said to have forward secrecy, if the compromise of a long-term private key at some point in the future does not compromise the security of communications made using that key in the past. Key transport via public key encryption does not have *forward secrecy*. To see why this is important, suppose you bulk encrypt a video stream and then encrypt the session key under the recipient's RSA public key. Then suppose that some time in the future, the recipient's RSA private key is compromised. At that point your video stream is also compromised, assuming the attacker recorded this at the time it was transmitted.

In addition using key transport implies that the recipient trusts the sender to be able to generate, in a sensible way, the session key. Sometimes the recipient may wish to contribute some randomness of their own to the session key. However, this can only be done if both parties are online at the same moment in time. Key transport is more suited to the case where only the sender is online, as in applications like email for example.

The key distribution problem was solved in the same seminal paper by Diffie and Hellman as that in which they introduced public key cryptography. Their protocol for key distribution, called Diffie–Hellman Key Exchange, allows two parties to agree a secret key over an insecure channel without having met before. Its security is based on the discrete logarithm problem in a finite abelian group G.

In the original paper the group is taken to be $G = \mathbb{F}_p^*$, but now more efficient versions can be produced by taking G to be an elliptic curve group, where the protocol is called EC-DH. The basic message flows for the Diffie–Hellman protocol are given in the following diagram:

$$\begin{array}{ccc} \text{Alice} & & \text{Bob} \\ a & g^a \longrightarrow & g^a \\ g^b & \longleftarrow g^b & b \end{array}$$

The two parties each have their own ephemeral secrets a and b. From these secrets both parties can agree on the same secret session key:

- Alice can compute $K = (g^b)^a$, since she knows a and was sent g^b by Bob,
- Bob can also compute $K = (g^a)^b$, since he knows b and was sent g^a by Alice.

Eve, the attacker, can see the messages

$$g^a \text{ and } g^b$$

and then needs to recover the secret key

$$K = g^{ab}$$

which is exactly the Diffie–Hellman problem considered in Chapter 11. Hence, the security of the above protocol rests not on the difficulty of solving the discrete logarithm problem, DLP, but on the difficulty of solving the Diffie–Hellman problem, DHP. Recall that it may be the case that it is easier to solve the DHP than the DLP, although no one believes this to be true for the groups that are currently used in real-life protocols.

Notice that the Diffie–Hellman protocol can be performed both online (in which case both parties contribute to the randomness in the shared session key) or offline, where one of the parties uses a long-term key of the form g^a instead of an ephemeral key. Hence, the Diffie–Hellman protocol can be used as a key exchange or as a key transport protocol.

The following is a very small example, in real life one takes $p \approx 2^{1024}$, but for our purposes we let the domain parameters be given by

$$p = 2\,147\,483\,659 \text{ and } g = 2.$$

Then the following diagram indicates a possible message flow for the Diffie–Hellman protocol:

$$\begin{array}{lcl} \text{Alice} & & \text{Bob} \\ a = 12\,345 & & b = 654\,323 \\ A = g^a = 428\,647\,416 & \longrightarrow & A = 428\,647\,416 \\ B = 450\,904\,856 & \longleftarrow & B = g^b = 450\,904\,856 \end{array}$$

The shared secret key is then computed via

$$\begin{aligned} A^b &= 428\,647\,416^{654\,323} \pmod p, \\ &= 1\,333\,327\,162, \\ B^a &= 450\,904\,856^{12\,345} \pmod p, \\ &= 1\,333\,327\,162. \end{aligned}$$

Notice that group elements are transmitted in the protocol, hence when using a finite field such as $\mathbb{F}_p^*$ for the Diffie–Hellman protocol the communication costs are around 1024 bits in each direction, since it is prudent to choose $p \approx 2^{1024}$. However, when one uses an elliptic curve group $E(\mathbb{F}_q)$ one can choose $q \approx 2^{160}$, and so the communication costs are much less, namely around 160 bits in each direction. In addition the group exponentiation step for elliptic curves can be done more efficiently than that for finite prime fields.

As a baby example of EC-DH consider the elliptic curve

$$E : Y^2 = X^3 + X - 3$$

over the field $\mathbb{F}_{199}$. Let the base point be given by $G = (1, 76)$, then a possible message flow is given by

Alice		Bob
$a = 23$		$b = 86$
$A = [a]G = (2, 150)$	$\longrightarrow$	$A = (2, 150)$
$B = (123, 187)$	$\longleftarrow$	$B = [b]G = (123, 187)$

The shared secret key is then computed via

$$\begin{aligned} [b]A &= [86](2, 150) \\ &= (156, 75), \\ [a]B &= [23](123, 187) \\ &= (156, 75). \end{aligned}$$

The shared key is then taken to be the x-coordinate 156 of the computed point. In addition, instead of transmitting the points, we transmit the compression of the point, which results in a significant saving in bandwidth.

So we seem to have solved the key distribution problem. But there is an important problem: you need to be careful *who* you are agreeing a key with. Alice has no assurance that she is agreeing a key with Bob, which can lead to the following (wo)man in the middle attack:

Alice		Eve		Bob
a	$\longrightarrow$	g^a		
g^m	$\longleftarrow$	m		
g^{am}		g^{am}		
		n	$\longrightarrow$	g^n
		g^b	$\longleftarrow$	b
		g^{bn}		g^{bn}

In the man in the middle attack

- Alice agrees a key with Eve, thinking it is Bob she is agreeing a key with,
- Bob agrees a key with Eve, thinking it is Alice,
- Eve can now examine communications as they pass through her, she acts as a router. She does not alter the plaintext, so her actions go undetected.

So we can conclude that the Diffie–Hellman protocol on its own is not enough. For example how does Alice know who she is agreeing a key with? Is it Bob or Eve?

2. Digital Signature Schemes

One way around the man in the middle attack on the Diffie–Hellman protocol is for Alice to sign her message to Bob and Bob to sign his message to Alice. In that way both parties know who they are talking to. Signatures are an important concept of public key cryptography, they also were invented by Diffie and Hellman in the same 1976 paper, but the first practical system was due to Rivest, Shamir and Adleman.

The basic idea behind public key signatures is as follows:

$$\begin{aligned} \text{Message} + \text{Alice's private key} &= \text{Signature}, \\ \text{Message} + \text{Signature} + \text{Alice's public key} &= \text{YES/NO}. \end{aligned}$$

The above is called a signature scheme with appendix, since the signature is appended to the message before transmission, the message needs to be input into the signature verification procedure. Another variant is the signature scheme with message recovery, where the message is output by the signature verification procedure, as described in

$$\text{Message} + \text{Alice's private key} = \text{Signature},$$
$$\text{Signature} + \text{Alice's public key} = \text{YES/NO} + \text{Message}.$$

The main idea is that only Alice can sign a message, which could only come from her since only Alice has access to the private key. On the other hand anyone can verify Alice's signature, since everyone can have access to her public key.

The main problem is how are the public keys to be trusted? How do you know a certain public key is associated to a given entity? You may think a public key belongs to Alice, but it may belong to Eve. Eve can therefore sign cheques etc., and you would think they come from Alice. We seem to have the same key management problem as in symmetric systems, albeit now the problem is not one of keeping the keys secret, but making sure they are authentic. We shall return to this problem later.

A digital signature scheme consists more formally of two transformations:

- a secret signing transform S,
- a public verification transform V.

In the following discussion, we assume a signature with message recovery. For an appendix based scheme a simple change to the following will suffice.

Alice, sending a message m, calculates

$$s = S(m)$$

and then transmits s, where s is the digital signature on the message m. Note, we are not interested in keeping the message secret here, since we are only interested in knowing who it comes from. If confidentiality of the message is important then the signature s could be encrypted using, for example, the public key of the receiver.

The receiver of the signature s applies the public verification transform V to s. The output is then the message m and a bit v. The bit v indicates valid or invalid, i.e. whether the signature is good or not. If v is valid the recipient gets a guarantee of three important security properties:

- message integrity – the message has not been altered in transit,
- message origin – the message was sent by Alice,
- non-repudiation – Alice cannot claim she did not send the message.

Note, the first two of these properties are also provided by message authentication codes, MACs. However, the last property of non-repudiation is not provided by MACs and has important applications in e-commerce. To see why non-repudiation is so important, consider what would happen if you could sign a cheque and then say you did not sign it.

The RSA encryption algorithm is particularly interesting since it can be used directly as a signature algorithm with message recovery.

- The sender applies the RSA decryption transform to generate the signature, by taking the message and raising it to the private exponent d

$$s = m^d \pmod{N}.$$

- The receiver then applies the RSA encryption transform to recover the original message

$$m = s^e \pmod{N}.$$

But this raises the question as to how do we check for validity of the signature? If the original message is in a natural language such as English then one can verify that the extracted message is also in the same natural language. But this is not a solution for all possible messages. Hence one needs to add redundancy to the message.

One way of doing this is as follows. Suppose the message D is t bits long and the RSA modulus N is k bits long, with $t < k - 32$. We first pad D to the right by zeros to produce a string of length a multiple of eight. We then add $(k - t)/8$ bytes to the left of D to produce a byte-string

$$m = 00\|01\|FF\|FF \ldots \|FF\|00\|D.$$

The signature is then computed via

$$m^d \pmod{N}.$$

When verifying the signature we ensure that the recovered value of m has the correct padding.

But not all messages will be so short so as to fit into the above method. Hence, naively to apply the RSA signature algorithm to a long message m we need to break it into blocks and sign each block in turn. This is very time consuming for long messages. Worse than this, we must add serial numbers and more redundancy to each message otherwise an attacker could delete parts of the long message without us knowing, just as happened when encrypting using a block cipher in ECB Mode. This problem arises because our signature model is one giving message recovery, i.e. the message is recovered from the signature and the verification process. If we used a system using a signature scheme with appendix then we could produce a hash of the message to be signed and then just sign the hash.

3. The Use of Hash Functions In Signature Schemes

Using a cryptographic hash function h, such as those described in Chapter 10, it is possible to make RSA into a signature scheme without message recovery, which is much more efficient for long messages.

Suppose we are given a long message m for signing, we first compute $h(m)$ and then apply the RSA signing transform to $h(m)$, i.e. the signature is given by

$$s = h(m)^d \pmod{N}.$$

The signature and message are then transmitted together as the pair (m, s). Verifying a message/signature pair (m, s) generated using a hash function involves three steps.

- 'Encrypt' s using the RSA encryption function to recover h', i.e.
$$h' = s^e \pmod{N}.$$
- Compute $h(m)$ from m.
- Check whether $h' = h(m)$. If they agree accept the signature as valid, otherwise the signature should be rejected.

Actually in practice one also needs padding, as a hash function usually does not have output the whole of the integers modulo N. You could use the padding scheme given earlier when we discussed RSA with message recovery.

Recall that a cryptographic hash function needs to satisfy the following three properties:

(1) **Preimage Resistant:** It should be hard to find a message with a given hash value.
(2) **Collision Resistant:** It should be hard to find two messages with the same hash value.
(3) **Second Preimage Resistant:** Given one message it should be hard to find another message with the same hash value.

So why do we need to use a hash function which has these properties within the above signature scheme? We shall address these issues below:

3.1. Requirement for preimage resistance. The one-way property stops a cryptanalyst from cooking up a message with a given signature. For example, suppose we are using the RSA scheme with appendix just described but with a hash function which does not have the one-way property. We then have the following attack.

- Eve computes
$$h' = r^e \pmod{N}$$
for some random integer r.
- Eve also computes the pre-image of h' under h (recall we are assuming that h does not have the one-way property) i.e. Eve computes
$$m = h^{-1}(h').$$

Eve now has your signature (m, r) on the message m. Such a forgery is called an existential forgery in that the attacker may not have any control over the contents of the message on which they have obtained a digital signature.

3.2. Requirement for collision resistance. This is needed to avoid the following attack, which is performed by the legitimate signer.

- The signer chooses two messages m and m' with $h(m) = h(m')$.
- They sign m and output the signature (m, s).
- Later they repudiate this signature, saying it was really a signature on the message m'.

As a concrete example one could have that m is an electronic cheque for 1000 euros whilst m' is an electronic cheque for 10 euros.

3.3. Requirement for second preimage resistance. This property is needed to stop the following attack.

- An attacker obtains your signature (m, s) on a message m.
- The attacker finds another message m' with $h(m') = h(m)$.
- The attacker now has your signature (m', s) on the message m'.

Note, the security of any signature scheme which uses a cryptographic hash function, depends both on the security of the underlying hard mathematical problem, such as factoring or the discrete logarithm problem, and the security of the underlying hash function.

4. The Digital Signature Algorithm

We have already presented one digital signature scheme RSA. You may ask why do we need another one?

- What if someone breaks the RSA algorithm or finds that factoring is easy?
- RSA is not suited to some applications since signature generation is a very costly operation.
- RSA signatures are very large, some applications require smaller signature footprints.

One algorithm which addresses all of these concerns is the Digital Signature Algorithm, or DSA. One sometimes sees this referred to as the DSS, or Digital Signature Standard. Although originally designed to work in the group $\mathbb{F}_p^*$, where p is a large prime, it is now common to see it used using elliptic curves, in which case it is called EC-DSA. The elliptic curve variants of DSA run very fast and have smaller footprints and key sizes than almost all other signature algorithms.

We shall first describe the basic DSA algorithm as it applies to finite fields. In this variant the security is based on the difficulty of solving the discrete logarithm problem in the field $\mathbb{F}_p$.

DSA is a signature with appendix algorithm and the signature produced consists of two 160-bit integers r and s. The integer r is a function of a 160-bit random number k called the ephemeral key which changes with every message. The integer s is a function of

- the message,
- the signer's private key x,
- the integer r,
- the ephemeral key k.

Just as with the ElGamal encryption algorithm there are a number of domain parameters which are usually shared amongst a number of users. The DSA domain parameters are all public information and are much like those found in the ElGamal encryption algorithm. First a 160-bit prime number q is chosen, one then selects a large prime number p such that

- p has between 512 and 2048 bits,
- q divides $p-1$.

Finally we generate a random integer h less than p and compute

$$g = h^{(p-1)/q}.$$

If $g = 1$ then we pick a new value of h until we obtain $g \neq 1$. This ensures that g is an element of order q in the group $\mathbb{F}_p^*$, i.e.

$$g^q = 1 \pmod{p}.$$

After having decided on the domain parameters (p, q, g), each user generates their own private signing key x such that

$$0 < x < q.$$

The associated public key is y where

$$y = g^x \pmod{p}.$$

Notice that key generation for each user is much simpler than with RSA, since we only require a single modular exponentiation to generate the public key.

To sign a message m the user performs the following steps:

- Compute the hash value $h = H(m)$.
- Choose a random ephemeral key, $0 < k < q$.
- Compute
$$r = (g^k \pmod{p}) \pmod{q}.$$
- Compute
$$s = (h + xr)/k \pmod{q}.$$

The signature on m is then the pair (r, s), notice that this signature is therefore around 320 bits long.

To verify the signature (r, s) on the message m the verifier performs the following steps.

- Compute the hash value $h = H(m)$.
- $a = h/s \pmod{q}$.
- $b = r/s \pmod{q}$.
- Compute, where y is the public key of the sender,
$$v = (g^a y^b \pmod{p}) \pmod{q}.$$
- Accept the signature if and only if $v = r$.

As a baby example of DSA consider the following domain parameters

$$q = 13,\ p = 4q + 1 = 53 \text{ and } g = 16.$$

Suppose the public/private key pair of the user is given by $x = 3$ and

$$y = g^3 \pmod{p} = 15.$$

Now, if we wish to sign a message which has hash value $h = 5$, we first generate the ephemeral secret key $k = 2$ and then compute

$$r = (g^k \pmod{p}) \pmod{q} = 5,$$
$$s = (h + xr)/k \pmod{q} = 10.$$

To verify this signature the recipient computes

$$a = h/s \pmod{q} = 7,$$
$$b = r/s \pmod{q} = 7,$$
$$v = (g^a y^b \pmod{p}) \pmod{q} = 5.$$

Note $v = r$ and so the signature verifies correctly.

The DSA algorithm uses the subgroup of $\mathbb{F}_p^*$ of order q which is generated by g. Hence the discrete logarithm problem really is in the cyclic group $\langle g \rangle$ of order q. For security we insisted that we have

- $p > 2^{512}$, although $p > 2^{1024}$ may be more prudent, to avoid attacks via the Number Field Sieve,
- $q > 2^{160}$ to avoid attacks via the Baby-Step/Giant-Step method.

Hence, to achieve the rough equivalent of 80 bits of DES strength we need to operate on integers of roughly 1024 bits in length. This makes DSA slower even than RSA, since the DSA operation is more complicated than RSA. The verification operation in RSA requires only one exponentiation modulo a 1024-bit number, and even that is an exponentiation by a small number. For DSA, verification requires two exponentiations modulo a 1024-bit number, rather than one as in RSA. In addition the signing operation for DSA is more complicated due to the need to compute the value of s.

The main problem is that the DSA algorithm really only requires to work in a finite abelian group of size 2^{160}, but since the integers modulo p are susceptible to an attack from the Number Field Sieve we are required to work with group elements of 1024 bits in size. This produces a significant performance penalty.

Luckily we can generalize DSA to an arbitrary finite abelian group in which the discrete logarithm problem is hard. We can then use a group which provides a harder instance of the discrete logarithm problem, for example the group of points on an elliptic curve over a finite field.

We write $G = \langle g \rangle$ for a group generated by g, we assume that

- g has prime order $q > 2^{160}$,
- the discrete logarithm problem with respect to g is hard,
- there is a public function f such that

$$f : G \longrightarrow \mathbb{Z}/q\mathbb{Z}.$$

We summarize the different choices between DSA and EC-DSA in the following table:

Quantity	DSA	EC-DSA
G	$\langle g \rangle < \mathbb{F}_p^*$	$\langle P \rangle < E(\mathbb{F}_p)$
g	$g \in \mathbb{F}_p^*$	$P \in E(\mathbb{F}_p)$
y	g^x	$[x]P$
f	$\cdot \pmod{q}$	x-coord$(P) \pmod{q}$

For this generalized form of DSA each user again generates a secret signing key, x. The public key is again give by y where

$$y = g^x.$$

Signatures are computed via the steps

- Compute the hash value $h = H(m)$.
- Chooses a random ephemeral key, $0 < k < q$.
- Compute
$$r = f(g^k).$$
- Compute
$$s = (h + xr)/k \pmod{q}.$$

The signature on m is then the pair (r, s).

To verify the signature (r, s) on the message m the verifier performs the following steps.

- Compute the hash value $h = H(m)$.
- $a = h/s \pmod{q}$.
- $b = r/s \pmod{q}$.
- Compute, where y is the public key of the sender,
$$v = f(g^a y^b).$$
- Accept the signature if and only if $v = r$.

You should compare this signature and verification algorithm with that given earlier for DSA and spot where they differ. When used for EC-DSA the above generalization is written additively.

As a baby example of EC-DSA take the following elliptic curve

$$Y^2 = X^3 + X + 3,$$

over the field $\mathbb{F}_{199}$. The number of elements in $E(\mathbb{F}_{199})$ is equal to $q = 197$ which is a prime, the elliptic curve group is therefore cyclic and as a generator we can take

$$P = (1, 76).$$

As a private key let us take $x = 29$, and so the associated public key is given by

$$Y = [x]P = [29](1, 76) = (113, 191).$$

Suppose the holder of this public key wishes to sign a message with hash value $H(m)$ equal to 68. They first produce a random ephemeral key, which we shall take to be $k = 153$ and compute

$$\begin{aligned} r &= x\text{-coord}\,([k]P) \\ &= x\text{-coord}\,([153](1, 76)) \\ &= x\text{-coord}\,((185, 35)) \\ &= 185. \end{aligned}$$

Now they compute

$$\begin{aligned} s &= (H(m) + x \cdot r)/k \pmod{q} \\ &= (68 + 29 \cdot 185)/153 \pmod{197} \\ &= 78. \end{aligned}$$

The signature is then the pair $(r, s) = (185, 78)$.

To verify this signature we compute

$$\begin{aligned} a &= H(m)/s \pmod{q} \\ &= 68/78 \pmod{197} \\ &= 112, \\ b &= r/s \pmod{q} \\ &= 185/78 \pmod{197} \\ &= 15. \end{aligned}$$

We then compute

$$\begin{aligned} Z &= [a]P + [b]Y \\ &= [112](1,76) + [15](113,191) \\ &= (111,60) + (122,140) \\ &= (185,35). \end{aligned}$$

The signature now verifies since we have

$$r = 185 = x\text{-coord}(Z).$$

5. Schnorr Signatures

There are many variants of signature schemes based on discrete logarithms. A particularly interesting one is that of Schnorr signatures. We present the algorithm in the general case and allow the reader to work out the differences between the elliptic curve and finite field variants.

Suppose G is a public finite abelian group generated by an element g of prime order q. The public/private key pairs are just the same as in DSA, namely

- The private key is an integer x in the range $0 < x < q$.
- The public key is the element
$$y = g^x.$$

To sign a message m using the Schnorr signature algorithm we:

(1) Choose an ephemeral key k in the range $0 < k < q$.
(2) Compute the associated ephemeral public key
$$r = g^k.$$
(3) Compute $e = h(m\|r)$. Notice how the hash function depends both on the message and the ephemeral public key.
(4) Compute
$$s = k + x \cdot e \pmod{q}.$$

The signature is then given by the pair (e, s).

The verification step is very simple, we first compute

$$r = g^s y^{-e}.$$

The signature is accepted if and only if $e = h(m\|r)$.

As an example of Schnorr signatures in a finite field we take the domain parameters

$$q = 101, p = 607 \text{ and } g = 601.$$

As the public/private key pair we assume $x = 3$ and

$$y = g^x \pmod{p} = 391.$$

Then to sign a message we generate an ephemeral key $k = 65$ and compute

$$r = g^k \pmod{p} = 223.$$

We now need to compute the hash value

$$e = h(m\|r) \pmod{q}.$$

Let us assume that we compute $e = 93$, then the second component of the signature is given by

$$\begin{aligned} s &= k + x \cdot e \pmod{q} \\ &= 65 + 3 \cdot 93 \pmod{101} \\ &= 41. \end{aligned}$$

In a later chapter we shall see that Schnorr signatures are able to be proved to be secure, assuming that discrete logarithms are hard to compute, whereas no proof of security is known for DSA signatures.

Schnorr signatures have been suggested to be used for challenge response mechanisms in smart cards since the response part of the signature (the value of s) is particularly easy to evaluate since it only requires the computation of a single modular multiplication and a single modular addition. No matter what group we choose this final phase only requires arithmetic modulo a relatively small prime number.

To see how one uses Schnorr signatures in a challenge response situation we give the following scenario. A smart card wishes to authenticate you to a building or ATM machine. The card reader has a copy of your public key y, whilst the card has a copy of your private key x. Whilst you are walking around the card is generating commitments, which are ephemeral public keys of the form

$$r = g^k.$$

When you place your card into the card reader the card transmits to the reader the value of one of these precomputed commitments. The card reader then responds with a challenge message e. Your card then only needs to compute

$$s = k + xe \pmod{q},$$

and transmit it to the reader which then verifies the 'signature', by checking that

$$g^s = ry^e.$$

Notice that the only online computation needed by the card is the computation of the value of e and s, which are both easy to perform.

In more detail, if we let C denote the card and R denote the card reader then we have

$$\begin{aligned} &C \longrightarrow R : r = g^k, \\ &R \longrightarrow C : e, \\ &C \longrightarrow R : s = k + xe \pmod{q}. \end{aligned}$$

The point of the initial commitment is to stop either the challenge being concocted so as to reveal your private key, or your response being concocted so as to fool the reader. A three-phase protocol consisting of

$$\text{commitment} \longrightarrow \text{challenge} \longrightarrow \text{response}$$

is a common form of authentication protocols, we shall see more protocols of this nature when we discuss zero-knowledge proofs in Chapter 25.

6. Nyberg–Rueppel Signatures

What happens when we want to sign a general message which is itself quite short. It may turn out that the signature could be longer than the message. Recall that RSA can be used either as a scheme with appendix or as a scheme with message recovery. So far none of our discrete logarithm based schemes can be used with message recovery. We shall now give an example scheme which does have the message recovery property, called the Nyberg–Rueppel signature scheme, which is based on discrete logarithms in some public finite abelian group G.

All signature schemes with message recovery require a public redundancy function R. This function maps actual messages over to the data which is actually signed. This acts rather like a hash function does in the schemes based on signatures with appendix. However, unlike a hash function the redundancy function must be easy to invert. As a simple example we could take R to be the function

$$R : \begin{cases} \{0,1\}^{n/2} \longrightarrow \{0,1\}^n \\ m \longmapsto m\|m. \end{cases}$$

We assume that the codomain of R can be embedded into the group G. In our description we shall use the integers modulo p, i.e. $G = \mathbb{F}_p^*$, and as usual we assume that a large prime q divides $p-1$ and that g is a generator of the subgroup of order q.

Once again the public/private key pair is given as a discrete logarithm problem

$$(y = g^x, x).$$

Nyberg–Rueppel signatures are then produced as follows:

(1) Select a random $k \in \mathbb{Z}/q\mathbb{Z}$ and compute
$$r = g^k \pmod{p}.$$
(2) Compute
$$e = R(m) \cdot r \pmod{p}.$$
(3) Compute
$$s = x \cdot e + k \pmod{q}.$$

The signature is then the pair (e, s). From this pair, which is a group element and an integer modulo q, we need to

- verify that the signature comes from the user with public key y,
- recover the message m from the pair (e, s).

Verification for a Nyberg–Rueppel signature takes the signature (e, s) and the sender's public key $y = g^x$ and then computes

(1) Set
$$u_1 = g^s y^{-e} = g^{s-ex} = g^k \pmod{p}.$$
(2) Now compute
$$u_2 = e/u_1 \pmod{p}.$$
(3) Verify that u_2 lies in the range of the redundancy function, e.g. we must have
$$u_2 = R(m) = m\|m.$$
If this does not hold then reject the signature.
(4) Recover the message $m = R^{-1}(u_2)$ and accept the signature.

As an example we take the domain parameters

$$q = 101, p = 607 \text{ and } g = 601.$$

As the public/private key pair we assume $x = 3$ and

$$y = g^x \pmod{p} = 391.$$

To sign the message $m = 12$, where m must lie in $[0, \ldots, 15]$, we compute an ephemeral key $k = 45$ and

$$r = g^k \pmod{p} = 143.$$

Suppose

$$R(m) = m + 2^4 \cdot m$$

then we have $R(m) = 204$. We then compute

$$e = R(m) \cdot r \pmod{p} = 36,$$
$$s = x \cdot e + k \pmod{q} = 52.$$

The signature is then the pair $(e, s) = (36, 52)$. We now show how this signature is verified and the message recovered. We first compute

$$u_1 = g^s y^{-e} = 143.$$

Notice how the verifier has computed u_1 to be the same as the value of r computed by the signer. The verifier now computes

$$u_2 = e/u_1 \pmod{p} = 204.$$

The verifier now checks that $u_2 = 204$ is of the form

$$m + 2^4 m$$

for some value of $m \in [0, \ldots, 15]$. We see that u_2 is of this form and so the signature is valid. The message is then recovered by solving for m in

$$m + 2^4 m = 204,$$

from which we obtain $m = 12$.

7. Authenticated Key Agreement

Now we know how to perform digital signatures we can solve the problem with Diffie–Hellman key exchange. Recall that the man in the middle attack worked because each end did not know who they were talking to. We can now authenticate each end by requiring the parties to digitally sign their messages.

We will still obtain forward secrecy, since the long-term signing key is only used to provide authentication and is not used to perform a key transport operation.

We also have two choices of Diffie–Hellman protocol, namely one based on the discrete logarithms in a finite field DH and one based on elliptic curves EC-DH. There are also at least three possible signing algorithms RSA, DSA and EC-DSA. Assuming security sizes of 1024 bits for RSA, 1024 bits for the prime in DSA and 160 bits for the group order in both DSA and EC-DSA we obtain the following message sizes for our signed Diffie–Hellman protocol.

Algorithms	DH size	Signature size	Total size
DH+DSA	1024	320	1344
DH+RSA	1024	1024	2048
ECDH+RSA	160	1024	1184
ECDH+ECDSA	160	320	480

This is still an awfully large amount of overhead to simply agree what could be only a 128-bit session key.

To make the messages smaller Menezes, Qu and Vanstone invented the following protocol, called the MQV protocol based on the DLOG problem in a group G generated by g. One can use this protocol either in finite fields or in elliptic curves to obtain authenticated key exchange with message size of

Protocol	Message size
DL-MQV	1024
EC-MQV	160

Thus the MQV protocol gives us a considerable saving on the earlier message sizes. The protocol works by assuming that both parties, Alice and Bob, generate first a long-term public/private key pair which we shall denote by

$$(A = g^a, a) \text{ and } (B = g^b, b).$$

We shall assume that Bob knows that A is the authentic public key belonging to Alice and that Alice knows that B is the authentic public key belonging to Bob. This authentication of the public keys can be ensured by using some form of public key certification, described in a later chapter.

Assume Alice and Bob now want to agree on a secret session key to which they both contribute a random nonce. The use of the nonces provides them with forward secrecy and means that neither party has to trust the other in producing their session keys. So Alice and Bob now generate a public/private ephemeral key pair each

$$(C = g^c, c) \text{ and } (D = g^d, d).$$

They then exchange C and D. These are the only message flows in the MQV protocol, namely

$$\begin{aligned} \text{Alice} &\longrightarrow \text{Bob} : g^c, \\ \text{Bob} &\longrightarrow \text{Alice} : g^d. \end{aligned}$$

Hence, to some extent this looks like a standard Diffie–Hellman protocol with no signing. However, the trick is that the final session key will also depend on the long-term public keys A and B.

Assume you are Alice, so you know

$$A, B, C, D, a \text{ and } c.$$

Let l denote half the bit size of the order of the group G, for example if we are using a group with order $q \approx 2^{160}$ then we set $l = 160/2 = 80$. To determine the session key, Alice now computes

(1) Convert C to an integer i.
(2) Put $s_A = (i \pmod{2^l}) + 2^l$.
(3) Convert D to an integer j.
(4) Put $t_A = (j \pmod{2^l}) + 2^l$.
(5) Put $h_A = c + s_A a$.
(6) Put $P_A = (DB^{t_A})^{h_A}$.

Bob runs the same protocol but with the roles of the public and private keys swapped around in the obvious manner, namely

(1) Convert D to an integer i.
(2) Put $s_B = (i \pmod{2^l}) + 2^l$.
(3) Convert C to an integer j.
(4) Put $t_B = (j \pmod{2^l}) + 2^l$.
(5) Put $h_B = d + s_B b$.
(6) Put $P_B = (CA^{t_B})^{h_B}$.

Then $P_A = P_B$ is the shared secret. To see why the P_A computed by Alice and the P_B computed by Bob are the same we notice that the s_A and t_A seen by Alice, are swapped when seen by Bob, i.e. $s_A = t_B$ and $s_B = t_A$. Setting $\log(P)$ to be the discrete logarithm of P to the base g, we see

that

$$\begin{aligned}
\log(P_A) &= \log\left((DB^{t_A})^{h_A}\right) \\
&= (d + bt_A)h_A \\
&= d(c + s_A a) + bt_A(c + s_A a) \\
&= d(c + t_B a) + bs_B(c + t_B a) \\
&= c(d + s_B b) + at_B(d + s_B b) \\
&= (c + at_B)h_B \\
&= \log\left((CA^{t_B})^{h_B}\right) \\
&= \log(P_B).
\end{aligned}$$

Chapter Summary

- Diffie–Hellman key exchange can be used for two parties to agree on a secret key over an insecure channel. However, Diffie–Hellman is susceptible to the man in the middle attack and so requires some form of authentication of the communicating parties.
- Digital signatures provide authentication for both long-term and short-term purposes. They come in two variants either with message recovery or as a signature with appendix.
- The RSA encryption algorithm can be used in reverse to produce a public key signature scheme, but one needs to combine the RSA algorithm with a hash algorithm to obtain security for both short and long messages.
- DSA is a signature algorithm based on discrete logarithms, it has reduced bandwidth compared with RSA but is slower. EC-DSA is the elliptic curve variant of DSA, it also has the benefit of reduced bandwidth compared to DSA, but is more efficient than DSA.
- Other discrete logarithm based signature algorithms exist, all with different properties. Two we have looked at are Schnorr signatures and Nyberg–Rueppel signatures.
- Another way of using a key exchange scheme, without the need for digital signatures, is to use the MQV system. This has very small bandwidth requirements. It obtains implicit authentication of the agreed key, by combining the ephemeral exchanged key with the long-term static public key of each user, so as to obtain a new session key.

Further Reading

Details on more esoteric signature schemes such as one-time signatures, fail-stop signatures and undeniable signatures can be found in the books by Stinson and Schneier. These are also good places to look for further details about hash functions and message authentication codes, although by far the best reference in this area is *HAC*.

B. Schneier. *Applied Cryptography.* Wiley, 1996.

D. Stinson. *Cryptography: Theory and Practice.* CRC Press, 1995.

CHAPTER 15

Implementation Issues

Chapter Goals

- To show how exponentiation algorithms are implemented.
- To explain how modular arithmetic can be implemented efficiently on large numbers.
- To show how certain tricks can be used to speed up RSA and DSA operations.
- To show how finite fields of characteristic two can be implemented efficiently.

1. Introduction

In this chapter we examine how one actually implements cryptographic operations. We shall mainly be concerned with public key operations since those are the most complex to implement. For example, in RSA or DSA we have to perform a modular exponentiation with respect to a modulus of a thousand or more bits. This means we need to understand the implementation issues involved with both modular arithmetic and exponentiation algorithms.

There is another reason to focus on public key algorithms rather than private key ones: in general public key schemes run much slower than symmetric schemes. In fact they can be so slow that their use can seriously slow down networks and web servers. Hence, efficient implementation is crucial unless one is willing to pay a large performance penalty.

Since RSA is the easiest system to understand we will concentrate on this, although where special techniques exist for other schemes we will mention these as well. The chapter focuses on algorithms used in software, for hardware based algorithms one often uses different techniques entirely, but these alternative techniques are related to those used in software, so an understanding of software techniques is important.

2. Exponentiation Algorithms

So far in this book we have assumed that computing

$$a^b \pmod{c}$$

is an easy operation. We need this operation both in RSA and in systems based on discrete logarithms such as ElGamal encryption and DSA. In this section we concentrate on the exponentiation algorithms and assume that we can perform modular arithmetic efficiently. In a later section we shall discuss how to perform modular arithmetic.

As we have already stressed, the main operation in RSA and DSA is modular exponentiation

$$M = C^d \pmod{N}.$$

Firstly note it does not make sense to perform this via

- compute $R = C^d$,
- then compute $R \pmod{N}$.

To see this, consider

$$123^5 \pmod{511} = 28\,153\,056\,843 \pmod{511} = 359.$$

With this naive method one obtains a huge intermediate result, in our small case above this is

$$28\,153\,056\,843.$$

But in a real 1024-bit RSA multiplication this intermediate result would be in general

$$2^{1024} \cdot 1024$$

bits long. Such a number requires 10^{301} gigabytes simply to write down.

To stop this explosion in the size of any intermediate results we use the fact that we are working modulo N. But even here one needs to be careful, a naive algorithm would compute the above example by computing

$$\begin{aligned} x &= 123, \\ x^2 &= x \times x \pmod{511} = 310, \\ x^3 &= x \times x^2 \pmod{511} = 316, \\ x^4 &= x \times x^3 \pmod{511} = 32, \\ x^5 &= x \times x^4 \pmod{511} = 359. \end{aligned}$$

This requires four modular multiplications, which seems fine for our small example. But for a general RSA exponentiation by a 1024-bit exponent using this method would require around 2^{1024} modular multiplications. If each such multiplication could be done in under one millionth of a second we would still require 10^{294} years to perform an RSA decryption operation.

However, it is easy to see that, even in our small example, we can reduce the number of required multiplications by being a little more clever:

$$\begin{aligned} x &= 123, \\ x^2 &= x \times x \pmod{511} = 310, \\ x^4 &= x^2 \times x^2 \pmod{511} = 32, \\ x^5 &= x \times x^4 \pmod{511} = 359. \end{aligned}$$

Which only requires three modular multiplications rather than the previous four. To understand why we only require three modular multiplications notice that the exponent 5 has binary representation $0b101$ and so

- has bit length $t = 3$,
- has Hamming weight $h = 2$.

In the above example we required $1 = (h - 1)$ general multiplications and $2 = (t - 1)$ squarings. This fact holds in general, in that a modular exponentiation can be performed using

- $(h - 1)$ multiplications,
- $(t - 1)$ squarings,

where t is the bit length of the exponent and h is the Hamming weight. The average Hamming weight of an integer is $t/2$ so the number of multiplications and squarings is on average

$$t + t/2 - 1.$$

For a 1024-bit RSA modulus this means that the average number of modular multiplications needed to perform exponentiation by a 1024-bit exponent is at most 2048 and on average 1535.

The method used to achieve this improvement in performance is called the binary exponentiation method. This is because it works by reading each bit of the binary representation of the exponent

in turn, starting with the least significant bit and working up to the most significant bit. Algorithm 15.1 explains the method by computing

$$y = x^d \pmod{n}.$$

Algorithm 15.1: Binary exponentiation : Right-to-Left variant

```
y = 1
while d ≠ 0 do
    if (d mod 2) ≠= 0 then
        y = (y · x) mod n
        d = d − 1
    end
    d = d/2
    x = (x · x) mod n
end
```

The above binary exponentiation algorithm has a number of different names, some authors call it the *square and multiply* algorithm, since it proceeds by a sequence of squarings and multiplications, other authors call it the *indian exponentiation* algorithm. The above algorithm is called a *right to left* exponentiation algorithm since it processes the bits of d from the least significant bit up to the most significant bit.

Most of the time it is faster to perform a squaring operation than a general multiplication. Hence to reduce time even more one tries to reduce the total number of modular multiplications even further. This is done using window techniques which trade off precomputations (i.e. storage) against the time in the main loop.

To understand window methods better we first examine the binary exponentiation method again. But this time instead of a right to left variant, we process the exponent from the most significant bit first, thus producing a *left to right* binary exponentiation algorithm, see Algorithm 15.2. Again we assume we wish to compute

$$y = x^d \pmod{n}.$$

We first give a notation for the binary representation of the exponent

$$d = \sum_{i=0}^{t} d_i 2^i,$$

where $d_i \in \{0, 1\}$.

Algorithm 15.2: Binary exponentiation : Left-to-Right variant

```
y = 1
for i = t downto 0 do
    y = (y · y) mod n
    if d_i = 1 then y = (y · x) mod n
end
```

The above algorithm processes a single bit of the exponent on every iteration of the loop. Again the number of squarings is equal to t and the expected number of multiplications is equal to $t/2$.

In a window method we process w bits of the exponent at a time, as in Algorithm 15.3. We first precompute a table

$$x_i = x^i \pmod{n} \text{ for } i = 0, \ldots, 2^w - 1.$$

Then we write our exponent out, but this time taking w bits at a time,

$$d = \sum_{i=0}^{t/w} d_i 2^{iw},$$

where $d_i \in \{0, 1, 2, \ldots, 2^w - 1\}$.

Algorithm 15.3: Window exponentiation method

```
y = 1
for i = t/w downto 0 do
    for j = 0 to w − 1 do
        y = (y · y) mod n
    end
    j = d_i
    y = (y · x_j) mod n
end
```

Let us see this algorithm in action by computing

$$y = x^{215} \pmod{n}$$

with a window width of $w = 3$. We compute the d_i as

$$215 = 3 \cdot 2^6 + 2 \cdot 2^3 + 7.$$

Hence, our iteration to compute $x^{215} \pmod{n}$ computes in order

$$\begin{aligned} y &= 1, \\ y &= y \cdot x^3 = x^3, \\ y &= y^8 = x^{24}, \\ y &= y \cdot x^2 = x^{26}, \\ y &= y^8 = y^{208}, \\ y &= y \cdot x^7 = x^{215}. \end{aligned}$$

With a window method as above, we still perform t squarings but the number of multiplications reduces to t/w on average. One can do even better by adopting a sliding window method, where we now encode our exponent as

$$d = \sum_{i=0}^{l} d_i 2^{e_i}$$

where $d_i \in \{1, 3, 5, \ldots, 2^w - 1\}$ and $e_{i+1} - e_i \geq w$. By choosing only odd values for d_i and having a variable window width we achieve both decreased storage for the precomputed values and improved efficiency. After precomputing $x_i = x^i$ for $i = 1, 3, 5, \ldots, 2^w - 1$, we execute Algorithm 15.4.

The number of squarings remains again at t, but now the number of multiplications reduces to l, which is about $t/(w + 1)$ on average. In our example of computing $y = x^{215} \pmod{n}$ we have

$$215 = 2^7 + 5 \cdot 2^4 + 7,$$

Algorithm 15.4: Sliding window exponentiation

$y = 1$
for $i = l$ **downto** 0 **do**
  for $j = 0$ **to** $e_{i+1} - e_i - 1$ **do** $y = (y \cdot y) \mod n$
  $j = d_i$
  $y = (y \cdot x_j) \mod n$
end
for $j = 0$ **to** $e_0 - 1$ **do** $y = (y \cdot y) \mod n$

and so we execute the steps

$$\begin{aligned} y &= 1, \\ y &= y \cdot x = x^1, \\ y &= y^8 = x^8, \\ y &= y \cdot x^5 = x^{13}, \\ y &= y^{16} = x^{208}, \\ y &= y \cdot x^7 = x^{215}. \end{aligned}$$

Notice that all of the above window algorithms apply to exponentiation in any abelian group and not just the integers modulo n. Hence, we can use these algorithms to compute

$$\alpha^d$$

in a finite field or to compute

$$[d]P$$

on an elliptic curve, in the latter case we call this point multiplication rather than exponentiation.

An advantage with elliptic curve variants is that negation comes for free, in that given P it is easy to compute $-P$. This leads to the use of signed binary and signed window methods. We only present the signed window method. We precompute

$$P_i = [i]P \text{ for } i = 1, 3, 5, \ldots, 2^{w-1} - 1,$$

which requires only half the storage of the equivalent sliding window method or one quarter of the storage of the equivalent standard window method. We now write our multiplicand d as

$$d = \sum_{i=0}^{l} d_i 2^{e_i}$$

where $d_i \in \{\pm 1, \pm 3, \pm 5, \ldots, \pm(2^{w-1} - 1)\}$. The signed sliding window method for elliptic curves is then given by Algorithm 15.5

Algorithm 15.5: Signed sliding window method

$Q = 0$
for $i = l$ **downto** 0 **do**
  for $j = 0$ **to** $e_{i+1} - e_i - 1$ **do** $Q = [2]Q$
  $j = d_i$
  if $j > 0$ **then** $Q = Q + P_j$
  else $Q = Q - P_{-j}$
end
for $j = 0$ **to** $e_0 - 1$ **do** $Q = [2]Q$

3. Exponentiation in RSA

To speed up RSA exponentiation even more, a number of tricks are used which are special to RSA. The tricks used are different depending on whether we are performing an encryption/verification operation with the public key or a decryption/signing operation with the private key.

3.1. RSA Encryption/Verification. As already remarked in earlier chapters one often uses a small public exponent, for example $e = 3, 17$ or $65\,537$. The reason for these particular values is that they have small Hamming weight, in fact the smallest possible for an RSA public key, namely two. This means that the binary method, or any other exponentiation algorithm, will require only one general multiplication, but it will still need k squarings where k is the bit size of the public exponent. For example

$$\begin{aligned} M^3 &= M^2 \times M, \\ M^{17} &= M^{16} \times M, \\ &= (((M^2)^2)^2)^2 \times M. \end{aligned}$$

3.2. RSA Decryption/Signing. In the case of RSA decryption or signing the exponent will be a general 1000-bit number. Hence, we need some way of speeding up the computation. Luckily, since we are considering a private key operation we have access to the private key, and hence the factorization of the modulus,

$$N = p \cdot q.$$

Supposing we are decrypting a message, we therefore wish to compute

$$M = C^d \pmod{N}.$$

We speed up the calculation by first computing M modulo p and q:

$$\begin{aligned} M_p &= C^d \pmod{p} = C^{d \pmod{p-1}} \pmod{p}, \\ M_q &= C^d \pmod{q} = C^{d \pmod{q-1}} \pmod{q}. \end{aligned}$$

Since p and q are 512-bit numbers, the above calculation requires two exponentiations modulo 512-bit moduli and 512-bit exponents. This is faster than a single exponentiation modulo a 1024-bit number with a 1024-bit exponent.

But we now need to recover M from M_p and M_q, which is done using the Chinese Remainder Theorem as follows: We compute $T = p^{-1} \pmod{q}$ and store it with the private key. The message M can then be recovered from M_p and M_q via

- $u = (M_q - M_p)T \pmod{q}$,
- $M = M_p + up$.

This is why in Chapter 11 we said that when you generate a private key it is best to store p and q even though they are not mathematically needed.

4. Exponentiation in DSA

Recall that in DSA verification one needs to compute

$$r = g^a y^b.$$

This can be accomplished by first computing g^a and then y^b and then multiplying the results together. However, often it is easier to perform the two exponentiations simultaneously. There are a number of techniques to accomplish this, using various forms of window techniques etc. But all are essentially based on the following idea, called Shamir's trick.

We first compute the following look-up table

$$G_i = g^{i_0} y^{i_1}$$

where $i = (i_1, i_0)$ is the binary representation of i, for $i = 0, 1, 2, 3$. We then compute an exponent array from the two exponents a and b. This is a 2 by t array, where t is the maximum bit length of a and b. The rows of this array are the binary representation of the exponents a and b. We then let I_j, for $j = 1, \ldots, t$, denote the integers whose binary representation is given by the columns of this array. The exponentiation is then computed by setting $r = 1$ and computing

$$r = r^2 \cdot G_{I_j}$$

for $j = 1$ to t.

As an example suppose we wish to compute

$$r = g^{11} y^7,$$

hence we have $t = 4$. We precompute

$$G_0 = 1,\ G_1 = g,\ G_2 = y,\ G_3 = g \cdot y.$$

Since the binary representation of 11 and 7 is given by 1011 and 111, our exponent array is given by

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}.$$

The integers I_j then become

$$I_1 = 1,\ I_2 = 2,\ I_3 = 3,\ I_4 = 3.$$

Hence, the four steps of our algorithm become

$$\begin{aligned} r &= G_1 = g, \\ r &= r^2 \cdot G_2 = g^2 \cdot y, \\ r &= r^2 \cdot G_3 = (g^4 \cdot y^2) \cdot (g \cdot y) = g^5 \cdot y^3, \\ r &= r^2 \cdot G_3 = (g^{10} \cdot y^6) \cdot (g \cdot y) = g^{11} \cdot y^7. \end{aligned}$$

Note, elliptic curve analogues of Shamir's trick and its variants can be made which make use of signed representations for the exponent. We do not give these here, but leave them for the interested reader to investigate.

5. Multi-precision Arithmetic

We shall now explain how to perform modular arithmetic on 1024-bit numbers. We show how this is accomplished using modern processors, and why naive algorithms are usually replaced with a special technique due to Montgomery.

In a cryptographic application it is common to focus on a fixed length for the integers in use, for example 1024 bits in an RSA/DSA implementation or 200 bits for an ECC implementation. This leads to different programming choices than when one implements a general purpose multi-precision arithmetic library. For example one no longer needs to worry so much about dynamic memory allocation, and one can now concentrate on particular performance enhancements for the integer sizes one is dealing with.

It is common to represent all integers in little-wordian format. This means that if a large integer is held in memory locations, $x_0, x_1, \ldots, x_n$, then x_0 is the least significant word and x_n is the most

significant word. For a 32-bit machine and 64-bit numbers we would represent x and y as $[x_0, x_1]$ and $[y_0, y_1]$ where

$$x = x_1 2^{32} + x_0,$$
$$y = y_1 2^{32} + y_0.$$

5.1. Addition. Most modern processors have a carry flag which is set by any overflow from an addition operation. Also most have a special instruction, usually called something like **addc**, which adds two integers together and adds on the contents of the carry flag. So if we wish to add our two 64-bit integers given earlier then we need to compute

$$z = x + y = z_2 2^{64} + z_1 2^{32} + z_0.$$

The values of z_0, z_1 and z_2 are then computed via

```
z0 <- add   x0,y0
z1 <- addc  x1,y1
z2 <- addc  0,0
```

Note that the value held in z_2 is at most one, so the value of z could be a 65-bit integer. The above technique for adding two 64-bit integers can clearly be scaled to adding integers of any fixed length, and can also be made to work for subtraction of large integers.

5.2. School-book Multiplication. We now turn to the next simplest arithmetic operation, after addition and subtraction, namely multiplication. Notice that two 32 -bit words multiply together to form a 64-bit result, and so most modern processors have an instruction which will do this operation.

$$w_1 \cdot w_2 = (High, Low) = (H(w_1 \cdot w_2), L(w_1 \cdot w_2)).$$

When we use school-book long multiplication, for our two 64-bit numbers, we obtain something like

$$\begin{array}{rrrr} & & x_1 & x_0 \\ & \times & y_1 & y_0 \\ \hline & & H(x_0 \cdot y_0) & L(x_0 \cdot y_0) \\ & H(x_0 \cdot y_1) & L(x_0 \cdot y_1) & \\ & H(x_1 \cdot y_0) & L(x_1 \cdot y_0) & \\ H(x_1 \cdot y_1) & L(x_1 \cdot y_1) & & \end{array}$$

Then we add up the four rows to get the answer, remembering we need to take care of the carries. This then becomes, for

$$z = x \cdot y,$$

something like the following pseudo-code

```
(z1,z0) <- mul x0,y0
(z3,z2) <- mul x1,y1
(h,l)   <- mul x1,y0
z1      <- add z1,l
z2      <- addc z2,h
z3      <- addc z3,0
(h,l)   <- mul x0,y1
z1      <- add z1,l
z2      <- addc z2,h
z3      <- addc z3,0
```

If n denotes the bit size of the integers we are operating on, the above technique for multiplying large integers together clearly requires $O(n^2)$ bit operations, whilst it requires $O(n)$ bit operations to add or subtract integers. It is a natural question as to whether one can multiply integers faster than $O(n^2)$.

5.3. Karatsuba Multiplication. One technique to speed up multiplication is called Karatsuba multiplication. Suppose we have two n-bit integers x and y that we wish to multiply. We write these integers as

$$x = x_0 + 2^{n/2}x_1,$$
$$y = y_0 + 2^{n/2}y_2,$$

where $0 \leq x_0, x_1, y_0, y_1 < 2^{n/2}$. We then multiply x and y by computing

$$A = x_0 \cdot y_0,$$
$$B = (x_0 + x_1) \cdot (y_0 + y_1),$$
$$C = x_1 \cdot y_1.$$

The product $x \cdot y$ is then given by

$$\begin{aligned} C2^n + (B - A - C)2^{n/2} + A &= x_1y_1 2^n + (x_1y_0 + x_0y_1)2^{n/2} + x_0y_0 \\ &= (x_0 + 2^{n/2}x_1) \cdot (y_0 + 2^{n/2}y_1) \\ &= x \cdot y. \end{aligned}$$

Hence, this multiplication technique to multiply two n-bit numbers requires three $n/2$-bit multiplications, two $n/2$-bit additions and three n-bit additions/subtractions. If we denote the cost of an n-bit multiplication by $M(n)$ and the cost of an n-bit addition/subtraction by $A(n)$ then this becomes

$$M(n) = 3M(n/2) + 2A(n/2) + 3A(n).$$

Now if we make the approximation that $A(n) \approx n$ then

$$M(n) \approx 3M(n/2) + 4n.$$

If the multiplication of the $n/2$-bit numbers is accomplished in a similar fashion then to obtain the final complexity of multiplication we need to solve the above recurrence relation to obtain

$$\begin{aligned} M(n) &\approx 9n^{\frac{\log(3)}{\log(2)}} \text{ as } n \longrightarrow \infty \\ &= 9n^{1.58}. \end{aligned}$$

So we obtain an algorithm with asymptotic complexity $O(n^{1.58})$. Karatsuba multiplication becomes faster than the $O(n^2)$ method for integers of sizes greater than a few hundred bits. However, one can do even better for very large integers since the fastest known multiplication algorithm takes time

$$O(n \log n \log \log n).$$

But neither this latter technique nor Karatsuba multiplication are used in many cryptographic applications. The reason for this will become apparent as we discuss integer division.

5.4. Division. After having looked at multiplication we are left with the division operation, which is the hardest of all the basic algorithms. Division is required in order to be able to compute the remainder on division, which is after all a basic operation in RSA. Given two large integers x and y we wish to be able to compute q and r such that

$$x = qy + r$$

where $0 \leq r < y$, such an operation is called a Euclidean division.

If we write our two integers x and y in the little-wordian format

$$x = (x_0, \ldots, x_n) \text{ and } y = (y_0, \ldots, y_t)$$

where the base for the representation is $b = 2^w$ then the Euclidean division can be performed by Algorithm 15.6. We let $t \ll_w v$ denote a large integer t shifted to the left by v words, in other words the result of multiplying t by b^v.

Algorithm 15.6: Euclidean division algorithm

$r = x$

/* Cope with the trivial case */
if $t > n$ **then**
  $q = 0$
  return
end
$q = 0, s = 0$

/* Normalise the divisor */
while $y_t < b/2$ **do**
  $y = 2y$
  $r = 2r$
  $s = s + 1$
end
if $r_{n+1} \neq 0$ **then** $n = n + 1$

/* Get the msw of the quotient */
while $r \geq (y \ll_w (n - t))$ **do**
  $q_{n-t} = q_{n-t} + 1$
  $r = r - (y \ll_w n - t)$
end

/* Deal with the rest */
for $i = n$ **to** $t + 1$ **do**
  if $r_i = y_t$ **then** $q_{i-t-1} = b - 1$
  else q_{i-t-1} =floor$((r_i b + r_{i-1})/y_t)$
  if $t \neq 0$ **then** $h_m = y_t b + y_{t-1}$
  else $h_m = y_t b$
  $h = q_{i-t-1} h_m$
  if $i \neq 1$ **then** $l = r_i b^2 + r_{i-1} b + r_{i-2}$
  else $l = r_i b^2 + r_{i-1} b$
  while $h > l$ **do**
    $q[i - t - 1] = q[i - t - 1] - 1$
    $h = h - h_m$
  end
  $r = r - (q_{i-t-1} y) \ll_w (i - t - 1)$
  if $r < 0$ **then**
    $r = r + (y \ll_w i - t - 1)$
    $q_{i-t-1} = q_{i-t-1} - 1$
  end
end

/* Renormalise */
for $i = 0$ **to** $s - 1$ **do** $r = r/2$

As one can see this is a complex operation, hence one should try and avoid divisions as much as possible.

5.5. Montgomery Arithmetic. That division is a complex operation means our cryptographic operations run very slowly if we use standard division operations as above. Recall that virtually all of our public key systems make use of arithmetic modulo another number. What we require is the ability to compute remainders (i.e. to perform modular arithmetic) without having to perform any costly division operations. This at first sight may seem a state of affairs which is impossible to reach, but it can be achieved using a special form of arithmetic called Montgomery arithmetic.

Montgomery arithmetic works by using an alternative representation of integers, called the Montgomery representation. Let us fix some notation, we let b denote 2 to the power of the word size of our computer, for example $b = 2^{32}$ or 2^{64}. To perform arithmetic modulo N we choose an integer R which satisfies

$$R = b^t > N.$$

Now instead of holding the value of the integer x in memory, we instead hold the value

$$x \cdot R \pmod{N}.$$

Again this is usually held in a little-wordian format. The value $x \cdot R \pmod{N}$ is called the Montgomery representation of the integer $x \pmod{N}$.

Adding two elements in Montgomery representation is easy. If

$$z = x + y \pmod{N}$$

then given $x \cdot R \pmod{N}$ and $y \cdot R \pmod{N}$ we need to compute $z \cdot R \pmod{N}$.

Algorithm 15.7: Addition in Montgomery representation

$zR = xR + yR$
if $zR \geq N$ **then** $zR = zR - N$

Let us take a simple example with

$$N = 1\,073\,741\,827,$$
$$b = R = 2^{32} = 4\,294\,967\,296.$$

The following is the map from the normal to Montgomery representation of the integers $1, 2$ and 3.

$$1 \longrightarrow 1 \cdot R \pmod{N} = 1\,073\,741\,815,$$
$$2 \longrightarrow 2 \cdot R \pmod{N} = 1\,073\,741\,803,$$
$$3 \longrightarrow 3 \cdot R \pmod{N} = 1\,073\,741\,791.$$

We can now verify that addition works since we have in the standard representation

$$1 + 2 = 3$$

whilst this is mirrored in the Montgomery representation as

$$1\,073\,741\,815 + 1\,073\,741\,803 = 1\,073\,741\,791 \pmod{N}.$$

Now we look at multiplication in Montgomery arithmetic. If we simply multiply two elements in Montgomery representation we will obtain

$$(xR) \cdot (yR) = xyR^2 \pmod{N}$$

but we want $xyR \pmod{N}$. Hence, we need to divide the result of the standard multiplication by R. Since R is a power of 2 we hope this should be easy.

The process of given y and computing

$$z = y/R \pmod{N}$$

given the earlier choice of R, is called Montgomery reduction. We first precompute the integer $q = 1/N \pmod{R}$, which is simple to perform with no divisions using the binary Euclidean algorithm. Then, performing a Montgomery reduction is done using Algorithm 15.8.

Algorithm 15.8: Montgomery reduction

$u = (-y \cdot q) \mod R$;
$z = (y + u \cdot N)/R$;
if $z \geq N$ **then** $z = z - N$;

Note that the reduction modulo R in the first line is easy, we compute $y \cdot q$ using standard algorithms, the reduction modulo R being achieved by truncating the result. This latter trick works since R is a power of b. The division by R in the second line can also be simply achieved, since $y + u \cdot N = 0 \pmod{R}$, we simply shift the result to the right by t words, again since $R = b^t$.

As an example we again take

$$N = 1\,073\,741\,827,$$
$$b = R = 2^{32} = 4\,294\,967\,296.$$

We wish to compute $2 \cdot 3$ in Montgomery representation. Recall

$$2 \longrightarrow 2 \cdot R \pmod{N} = 1\,073\,741\,803 = x,$$
$$3 \longrightarrow 3 \cdot R \pmod{N} = 1\,073\,741\,791 = y.$$

We then compute, using a standard multiplication algorithm that

$$w = x \cdot y = 1\,152\,921\,446\,624\,789\,173 = 2 \cdot 3 \cdot R^2.$$

We now need to pass this value of w into our technique for Montgomery reduction, so as to find the Montgomery representation of $x \cdot y$. We find

$$w = 1\,152\,921\,446\,624\,789\,173,$$
$$q = (1/N) \pmod{R} = 1\,789\,569\,707,$$
$$u = -w \cdot q \pmod{R} = 3\,221\,225\,241,$$
$$z = (w + u \cdot N)/R = 1\,073\,741\,755.$$

So the multiplication of x and y in Montgomery arithmetic should be

$$1\,073\,741\,755.$$

We can check that this is the correct value by computing

$$6 \cdot R \pmod{N} = 1\,073\,741\,755.$$

Hence, we see that Montgomery arithmetic allows us to add and multiply integers modulo an integer N without the need for costly division algorithms.

Our above method for Montgomery reduction requires two full multi-precision multiplications. So to multiply two numbers in Montgomery arithmetic we require three full multi-precision multiplications. If we are multiplying 1024-bit numbers, this means the intermediate results can grow to be 2048-bit numbers. We would like to do better, and we can.

Suppose y is given in little-wordian format

$$y = (y_0, y_1, \ldots, y_{2t-2}, y_{2t-1}).$$

Then a better way to perform Montgomery reduction is to first precompute

$$N' = -1/N \pmod{b}$$

which is easy and only requires operations on word-sized quantities, and then to execute Algorithm 15.9

Algorithm 15.9: Word oriented Montgomery reduction

$z = y$
for $i = 0$ **to** $t - 1$ **do**
  $u = (z_i \cdot N') \mod b$
  $z = z + u \cdot N$
  $z = z \cdot b$
end
$z = z/R$
if $z \geq N$ **then** $z = z - N$

Note, since we are reducing modulo b in the first line of the for loop we can execute this initial multiplication using a simple word multiplication algorithm. The second step of the for loop requires a shift by one word (to multiply by b) and a single *word* $\times$ *bigint* multiply. Hence, we have reduced the need for large intermediate results in the Montgomery reduction step.

We can also interleave the multiplication with the reduction to perform a single loop to produce

$$Z = XY/R \pmod{N}.$$

So if $X = xR$ and $Y = yR$ this will produce

$$Z = (xy)R.$$

This procedure is called Montgomery multiplication and allows us to perform a multiplication in Montgomery arithmetic without the need for larger integers, as in Algorithm 15.10.

Algorithm 15.10: Montgomery multiplication

$Z = 0$
for $i = 0$ **to** $t - 1$ **do**
  $u = ((z_0 + X_i \cdot Y_0) \cdot N') \mod b$
  $Z = (Z + X_i \cdot Y + u \cdot N)/b$
end
if $Z \geq N$ **then** $Z = Z - N$

Whilst Montgomery multiplication has complexity $O(n^2)$ as opposed to the $O(n^{1.58})$ of Karatsuba multiplication, it is still preferable to use Montgomery arithmetic since it deals more efficiently with modular arithmetic.

6. Finite Field Arithmetic

Apart from the integers modulo a large prime p the other type of finite field used in cryptography are those based on fields of characteristic two. These occur in the Rijndael algorithm and in certain elliptic curve systems. In Rijndael the field is so small that one can use look-up tables or special circuits to perform the basic arithmetic tasks, so in this section we shall concentrate on fields of large degree over $\mathbb{F}_2$, like those used with elliptic curves. In addition we shall concern ourselves with software implementations only. Fields of characteristic two can have special types of hardware implementations based on things called optimal normal bases, but we shall not concern ourselves with these.

Recall that to define a finite field of characteristic two we first pick an irreducible polynomial $f(x)$ over $\mathbb{F}_2$ of degree n. The field is defined to be

$$\mathbb{F}_{2^n} = \mathbb{F}_2[x]/f(x),$$

i.e. we look at binary polynomials modulo $f(x)$. Elements of this field are usually represented as bit strings, which represent a binary polynomial. For example the bit string

$$101010111$$

represents the polynomial

$$x^8 + x^6 + x^4 + x^2 + x + 1.$$

Addition and subtraction of elements in $\mathbb{F}_{2^n}$ is accomplished by simply performing a bitwise XOR between the two bitstrings. Hence, the difficult tasks are multiplication and division.

It turns out that division, although slower than multiplication, is easier to describe, so we start with division. To compute

$$\alpha/\beta,$$

where $\alpha, \beta \in \mathbb{F}_{2^n}$, we first compute

$$\beta^{-1}$$

and then perform the multiplication

$$\alpha \cdot \beta^{-1}.$$

So division is reduced to multiplication and the computation of β^{-1}. One way of computing β^{-1} is to use Lagrange's Theorem which tells us for $\beta \neq 0$ that we have

$$\beta^{2^n-1} = 1.$$

But this means that

$$\beta \cdot \beta^{2^n-2} = 1,$$

or in other words

$$\beta^{-1} = \beta^{2^n-2} = \beta^{2(2^{n-1}-1)}.$$

Another way of computing β^{-1} is to use the binary Euclidean algorithm. We take the polynomial f and the polynomial b which represents β and then perform Algorithm 15.11, which is a version of the binary Euclidean algorithm, where $\mathrm{lsb}(b)$ refers to the least significant bit of b (in other words the coefficient of x^0),

We now turn to the multiplication operation. Unlike the case of integers modulo N or p, where we use a special method of Montgomery arithmetic, in characteristic two we have the opportunity to choose a polynomial $f(x)$ which has 'nice' properties. Any irreducible polynomials of degree n can be used to implement the finite field $\mathbb{F}_{2^n}$, we just need to select the best one.

Almost always one chooses a value of $f(x)$ which is either a trinomial

$$f(x) = x^n + x^k + 1$$

or a pentanomial

$$f(x) = x^n + x^{k_3} + x^{k_2} + x^{k_1} + 1.$$

It turns out that for all fields of degree less than 10 000 we can always find such a trinomial or pentanomial to make the multiplication operation very efficient. Table 1 at the end of this chapter gives a list for all values of n between 2 and 500 of an example pentanomial or trinomial which defines the field $\mathbb{F}_{2^n}$. In all cases where a trinomial exists we give one, otherwise we present a pentanomial.

Now to perform a multiplication of α by β we first multiply the polynomials representing α and β together to form a polynomial $\gamma(x)$ of degree at most $2n-2$. Then we reduce this polynomial by taking the remainder on division by the polynomial $f(x)$.

Algorithm 15.11: Binary extended Euclidean algorithm for polynomials over $\mathbb{F}_2$

```
a = f
B = 0
D = 1
/* At least one of a and b now has a constant term on every
execution of the loop. */
while a ≠ 0 do
    while lsb(a) = 0 do
        a = a ≫ 1
        if lsb(B) ≠ 0 then  B = B ⊕ f
        B = B ≫ 1
    end
    while lsb(b) = 0 do
        b = b ≫ 1
        if lsb(D) ≠ 0 then  D = D ⊕ f
        D = D ≫ 1
    end
    /* Now both a and b have a constant term */
    if deg(a) ≥ deg(b) then
        a = a ⊕ b
        B = B ⊕ D
    else
        b = a ⊕ b
        D = D ⊕ B
    end
end
return D
```

We show how this remainder on division is efficiently performed for trinomials, and leave the pentanomial case for the reader. We write

$$\gamma(x) = \gamma_1(x)x^n + \gamma_0(x).$$

Hence, $\deg(\gamma_1(x)), \deg(\gamma_0(x)) \leq n-1$. We can then write

$$\gamma(x) \pmod{f(x)} = \gamma_0(x) + (x^k + 1)\gamma_1(x).$$

The right-hand side of this equation can be computed from the bit operations

$$\delta = \gamma_0 \oplus \gamma_1 \oplus (\gamma_1 \ll k).$$

Now δ, as a polynomial, will have degree at most $n-1+k$. So we need to carry out this procedure again by first writing

$$\delta(x) = \delta_1(x)x^n + \delta_0(x),$$

where $\deg(\delta_0(x)) \leq n-1$ and $\deg(\delta_1(x)) \leq k-1$. We then compute as before that γ is equivalent to

$$\delta_0 \oplus \delta_1 \oplus (\delta_1 \ll k).$$

This latter polynomial will have degree $\max(n-1, 2k-1)$, so if we may choose in our trinomial

$$k \leq n/2,$$

then Algorithm 15.12 will perform our division by remainder step. Let g denote the polynomial of degree $2n-2$ that we wish to reduce modulo f, where we assume a bit representation for these polynomials.

Algorithm 15.12: Reduction by a trinomial

$g_1 = g \gg n$
$g_0 = g[n-1 \ldots 0]$
$g = g_0 \oplus g_1 \oplus (g_1 \ll k)$
$g_1 = g \gg n$
$g_0 = g[n-1 \ldots 0]$
$g = g_0 \oplus g_1 \oplus (g_1 \ll k)$

So to complete our description of how to multiply elements in $\mathbb{F}_{2^n}$ we need to explain how to perform the multiplication of two binary polynomials of large degree $n-1$.

Again one can use a naive multiplication algorithm. Often however one uses a look-up table for polynomial multiplication of polynomials of degree less than eight, i.e. for operands which fit into one byte. Then multiplication of larger degree polynomials is reduced to multiplication of polynomials of degree less than eight by using a variant of the standard long multiplication algorithm from school. This algorithm will have complexity $O(n^2)$, where n is the degree of the polynomials involved.

Suppose we have a routine which uses a look-up table to multiply two binary polynomials of degree less than eight, returning a binary polynomial of degree less than sixteen. This function we denote by $Mult\_Tab(a, b)$ where a and b are 8-bit integers representing the input polynomials.

To perform a multiplication of two n-bit polynomials represented by two n-bit integers x and y we perform Algorithm 15.13, where $y \gg 8$ (resp. $y \ll 8$) represents shifting to the right (resp. left) by 8 bits.

Algorithm 15.13: Multiplication of two n-bit polynomials over $\mathbb{F}_2$

$i = 0, a = 0$
while $x \neq 0$ **do**
  $u = y, j = 0$
  while $u \neq 0$ **do**
    $w = Mult\_Tab(x\&255, u\&255)$
    $w = w \ll (8(i+j))$
    $a = a \oplus w$
    $u = u \gg 8, j = j + 1$
  end
  $x = x \gg 8, i = i + 1$
end
return (a)

Just as with integer multiplication one can use a divide and conquer technique based on Karatsuba multiplication, which again will have a complexity of $O(n^{1.58})$. Suppose the two polynomials we wish to multiply are given by

$$a = a_0 + x^{n/2} a_1,$$
$$b = b_0 + x^{n/2} b_2,$$

where a_0, a_1, b_0, b_1 are polynomials of degree less than $n/2$. We then multiply a and b by computing

$$A = a_0 \cdot b_0,$$
$$B = (a_0 + a_1) \cdot (b_0 + b_1),$$
$$C = a_1 \cdot b_1.$$

The product $a \cdot b$ is then given by

$$\begin{aligned} Cx^n + (B - A - C)x^{n/2} + A &= a_1 b_1 x^n + (a_1 b_0 + a_0 b_1)x^{n/2} + a_0 b_0 \\ &= (a_0 + x^{n/2} a_1) \cdot (b_0 + x^{n/2} b_2) \\ &= a \cdot b. \end{aligned}$$

Again to multiply a_0 and b_0 etc. we use the Karatsuba multiplication method recursively. Once we reduce to the case of multiplying two polynomials of degree less than eight we resort to using our look-up table to perform the polynomial multiplication. Unlike the integer case we now find that Karatsuba multiplication is more efficient than the school-book method even for polynomials of quite small degree, say $n \approx 100$.

One should note that squaring polynomials in characteristic two is particularly easy. Suppose we have a polynomial

$$a = a_0 + a_1 x + a_2 x^2 + a_3 x^3,$$

where $a_i = 0$ or 1. Then to square a we simply 'thin out' the coefficients as follows:

$$a^2 = a_0 + a_1 x^2 + a_2 x^4 + a_3 x^6.$$

This means that squaring an element in a finite field of characteristic two is very fast compared with a multiplication operation.

Chapter Summary

- Modular exponentiation, or exponentiation in any group, can be computed using the binary exponentiation method. Often it is more efficient to use a window based method, or to use a signed exponentiation method in the case of elliptic curves.
- For RSA special optimizations are performed. In the case of the public exponent we choose one which is both small and has very low Hamming weight. For the exponentiation by the private exponent we use knowledge of the prime factorization of the modulus and the Chinese Remainder Theorem.
- For DSA verification there is a method based on simultaneous exponentiation which is often more efficient than performing two single exponentiations and then combining the result.
- Modular arithmetic is usually implemented using the technique of Montgomery representation. This allows us to avoid costly division operations by replacing the division with simple shift operations. This however is at the expense of using a non-standard representation for the numbers.
- Finite fields in characteristic two can also be implemented efficiently. But now the modular reduction operation can be made simple by choosing a special polynomial $f(x)$. Inversion is also particular simple using a variant of the binary Euclidean algorithm, although often inversion is still 3–10 times slower than multiplication.

Further Reading

The standard reference work for the type of algorithms considered in this chapter is Volume 2 of Knuth. A more gentle introduction can be found in the book by Bach and Shallit, whilst for more algorithms one should consult the book by Cohen. The first chapter of Cohen gives a number

of lessons learnt in the development of the PARI/GP calculator which can be useful, whilst Bach and Shallit provides an extensive bibliography and associated commentary.

E. Bach and S. Shallit. *Algorithmic Number Theory, Volume 1: Efficient Algorithms.* MIT Press, 1996.

H. Cohen. *A Course in Computational Algebraic Number Theory.* Springer-Verlag, 1993.

D. Knuth. *The Art of Computing Programming, Volume 2 : Seminumerical Algorithms.* Addison-Wesley, 1975.

TABLE 1. Trinomials and pentanomials

n	$k/k_1, k_2, k_3$	n	$k/k_1, k_2, k_3$	n	$k/k_1, k_2, k_3$
2	1	3	1	4	1
5	2	6	1	7	1
8	7,3,2	9	1	10	3
11	2	12	3	13	4,3,1
14	5	15	1	16	5,3,1
17	3	18	3	19	5,2,1
20	3	21	2	22	1
23	5	24	8,3,2	25	3
26	4,3,1	27	5,2,1	28	1
29	2	30	1	31	3
32	7,3,2	33	10	34	7
35	2	36	9	37	6,4,1
38	6,5,1	39	4	40	5,4,3
41	3	42	7	43	6,4,3
44	5	45	4,3,1	46	1
47	5	48	11,5,1	49	9
50	4,3,2	51	6,3,1	52	3
53	6,2,1	54	9	55	7
56	7,4,2	57	4	58	19
59	7,4,2	60	1	61	5,2,1
62	29	63	1	64	11,2,1
65	32	66	3	67	5,2,1
68	33	69	6,5,2	70	37,34,33
71	35	72	36,35,33	73	42
74	35	75	35,34,32	76	38,33,32
77	38,33,32	78	41,37,32	79	40,36,32
80	45,39,32	81	35	82	43,35,32
83	39,33,32	84	35	85	35,34,32
86	49,39,32	87	46,34,32	88	45,35,32
89	38	90	35,34,32	91	41,33,32
92	37,33,32	93	35,34,32	94	43,33,32
95	41,33,32	96	57,38,32	97	33
98	63,35,32	99	42,33,32	100	37
101	40,34,32	102	37	103	72
104	43,33,32	105	37	106	73,33,32
107	54,33,32	108	33	109	34,33,32
110	33	111	49	112	73,51,32
113	37,33,32	114	69,33,32	115	53,33,32
116	48,33,32	117	78,33,32	118	33

119	38	120	41,35,32	121	35,34,32
122	39,34,32	123	42,33,32	124	37
125	79,33,32	126	49	127	63
128	55,33,32	129	46	130	61,33,32
131	43,33,32	132	44,33,32	133	46,33,32
134	57	135	39,33,32	136	35,33,32
137	35	138	57,33,32	139	38,33,32
140	45	141	85,35,32	142	71,33,32
143	36,33,32	144	59,33,32	145	52
146	71	147	49	148	61,33,32
149	64,34,32	150	53	151	39
152	35,33,32	153	71,33,32	154	109,33,32
155	62	156	57	157	47,33,32
158	76,33,32	159	34	160	79,33,32
161	39	162	63	163	48,34,32
164	42,33,32	165	35,33,32	166	37
167	35	168	134,33,32	169	34
170	105,35,32	171	125,34,32	172	81
173	71,33,32	174	57	175	57
176	79,37,32	177	88	178	87
179	80,33,32	180	33	181	46,33,32
182	81	183	56	184	121,39,32
185	41	186	79	187	37,33,32
188	46,33,32	189	37,34,32	190	47,33,32
191	51	192	147,33,32	193	73
194	87	195	50,34,32	196	33
197	38,33,32	198	65	199	34
200	57,35,32	201	59	202	55
203	68,33,32	204	99	205	94,33,32
206	37,33,32	207	43	208	119,34,32
209	45	210	49,35,32	211	175,33,32
212	105	213	75,33,32	214	73
215	51	216	115,34,32	217	45
218	71	219	54,33,32	220	33
221	63,33,32	222	102,33,32	223	33
224	39,33,32	225	32	226	59,34,32
227	81,33,32	228	113	229	64,35,32
230	50,33,32	231	34	232	191,33,32
233	74	234	103	235	34,33,32
236	50,33,32	237	80,34,32	238	73
239	36	240	177,35,32	241	70
242	95	243	143,34,32	244	111
245	87,33,32	246	62,33,32	247	82

248	155,33,32	249	35	250	103
251	130,33,32	252	33	253	46
254	85,33,32	255	52	256	91,33,32
257	41	258	71	259	113,33,32
260	35	261	89,34,32	262	86,33,32
263	93	264	179,33,32	265	42
266	47	267	42,33,32	268	61
269	207,33,32	270	53	271	58
272	165,35,32	273	53	274	67
275	81,33,32	276	63	277	91,33,32
278	70,33,32	279	38	280	242,33,32
281	93	282	35	283	53,33,32
284	53	285	50,33,32	286	69
287	71	288	111,33,32	289	36
290	81,33,32	291	168,33,32	292	37
293	94,33,32	294	33	295	48
296	87,33,32	297	83	298	61,33,32
299	147,33,32	300	45	301	83,33,32
302	41	303	36,33,32	304	203,33,32
305	102	306	66,33,32	307	46,33,32
308	40,33,32	309	107,33,32	310	93
311	78,33,32	312	87,33,32	313	79
314	79,33,32	315	132,33,32	316	63
317	36,34,32	318	45	319	36
320	135,34,32	321	41	322	67
323	56,33,32	324	51	325	46,33,32
326	65,33,32	327	34	328	195,37,32
329	50	330	99	331	172,33,32
332	89	333	43,34,32	334	43,33,32
335	113,33,32	336	267,33,32	337	55
338	86,35,32	339	72,33,32	340	45
341	126,33,32	342	125	343	75
344	135,34,32	345	37	346	63
347	56,33,32	348	103	349	182,34,32
350	53	351	34	352	147,34,32
353	69	354	99	355	43,33,32
356	112,33,32	357	76,34,32	358	57
359	68	360	323,33,32	361	56,33,32
362	63	363	74,33,32	364	67
365	303,33,32	366	38,33,32	367	171
368	283,34,32	369	91	370	139
371	116,33,32	372	111	373	299,33,32
374	42,33,32	375	64	376	227,33,32

377	41	378	43	379	44,33,32
380	47	381	107,34,32	382	81
383	90	384	295,34,32	385	51
386	83	387	162,33,32	388	159
389	275,33,32	390	49	391	37,33,32
392	71,33,32	393	62	394	135
395	301,33,32	396	51	397	161,34,32
398	122,33,32	399	49	400	191,33,32
401	152	402	171	403	79,33,32
404	65	405	182,33,32	406	141
407	71	408	267,33,32	409	87
410	87,33,32	411	122,33,32	412	147
413	199,33,32	414	53	415	102
416	287,38,32	417	107	418	199
419	200,33,32	420	45	421	191,33,32
422	149	423	104,33,32	424	213,34,32
425	42	426	63	427	62,33,32
428	105	429	83,33,32	430	62,33,32
431	120	432	287,34,32	433	33
434	55,33,32	435	236,33,32	436	165
437	40,34,32	438	65	439	49
440	63,33,32	441	35	442	119,33,32
443	221,33,32	444	81	445	146,33,32
446	105	447	73	448	83,33,32
449	134	450	47	451	406,33,32
452	97,33,32	453	87,33,32	454	128,33,32
455	38	456	67,34,32	457	61
458	203	459	68,33,32	460	61
461	194,35,32	462	73	463	93
464	143,33,32	465	59	466	143,33,32
467	156,33,32	468	33	469	116,34,32
470	149	471	119	472	47,33,32
473	200	474	191	475	134,33,32
476	129	477	150,33,32	478	121
479	104	480	169,35,32	481	138
482	48,35,32	483	288,33,32	484	105
485	267,33,32	486	81	487	94
488	79,33,32	489	83	490	219
491	61,33,32	492	50,33,32	493	266,33,32
494	137	495	76	496	43,33,32
497	78	498	155	499	40,33,32
500	75				

CHAPTER 16

Obtaining Authentic Public Keys

Chapter Goals

- To describe the notion of digital certificates.
- To explain the notion of a PKI.
- To examine different approaches such as X509, PGP and SPKI.
- To show how an implicit certificate scheme can operate.
- To explain how identity based cryptographic schemes operate.

1. Generalities on Digital Signatures

Digital signatures have a number of uses which go beyond the uses of handwritten signatures. For example we can use digital signatures to

- control access to data,
- allow users to authenticate themselves to a system,
- allow users to authenticate data,
- sign 'real' documents.

Each application has a different type of data being bound, a different length of the lifetime of the data to be signed, different types of principals performing the signing and verifying and different awareness of the data being bound.

For example an interbank payment need only contain the two account numbers and the amount. It needs to be signed by the payee and verified only by the computer which will carry out the transfer. The lifetime of the signature is only until the accounts are reconciled, for example when the account statements are sent to the customers and a suitable period has elapsed to allow the customers to complain of any error.

As another example consider a challenge response authentication mechanism. Here the user, to authenticate itself to the device, signs a challenge provided by the device. The lifetime of the signature may only be a few seconds. The user of course assumes that the challenge is random and is not a hash of an interbank payment. Hence, it is probably prudent that we use different keys for our authentication tokens and our banking applications.

As a final example consider a digital will or a mortgage contract. The length of time that this signature must remain valid may (hopefully in the case of a will) be many years. Hence, the security requirements for long-term legal documents will be very different from those of an authentication token.

You need to remember however that digital signatures are unlike handwritten signatures in that they are

- NOT necessarily on a document: Any piece of digital stuff can be signed.
- NOT transferable to other documents: Unlike a handwritten signature, a digital signature is different on each document.

- NOT modifiable after they are made: One cannot alter the document and still have the digital signature remaining valid.
- NOT produced by a person: A digital signature is never produced by a person, unless the signature scheme is very simple (and weak) or the person is a mathematical genius.

All they do is bind knowledge of an unrevealed private key to a particular piece of data.

2. Digital Certificates and PKI

When using a symmetric key system we assume we do not have to worry about which key belongs to which principle. It is tacitly assumed, see for example the chapter dealing with symmetric key agreement protocols and the BAN logic, that if Alice holds a long-term secret key K_{ab} which she thinks is shared with Bob, then Bob really does have a copy of the same key. This assurance is often achieved using a trusted physical means of long-term key distribution, using for example armed couriers.

In a public key system the issues are different. Alice may have a public key which she thinks is associated with Bob, but we usually do not assume that Alice is 100 percent certain that it really belongs to Bob. This is because we do not, in the public key model, assume a physically secure key distribution system. After all, that was the point of public key cryptography in the first place: to make key management easier. Alice may have obtained the public key she thinks belongs to Bob from Bob's web page, but how does she know the web page has not been spoofed?

The process of linking a public key to an entity or principal, be it a person, machine or process, is called binding. One way of binding, common in many applications where the principal really does need to be present, is by using a physical token such as a smart card. Possession of the token, and knowledge of any PIN/password needed to unlock the token, is assumed to be equivalent to being the designated entity. This solution has a number of problems associated with it, since cards can be lost or stolen, which is why we protect them using a PIN (or in more important applications by using biometrics). The major problem is that most entities are non-human, they are computers and computers do not carry cards. In addition many public key protocols are performed over networks where physical presence of the principal (if it is human) is not something one can test.

Hence, some form of binding is needed which can be used in a variety of very different applications. The main binding tool in use today is the digital certificate. In this a special trusted third party, or TTP, called a certificate authority, or CA, is used to vouch for the validity of the public keys.

A CA based system works as follows:

- All users have a trusted copy of the public key of the CA. For example these come embedded in your browser when you buy your computer, and you 'of course' trust the vendor of the computer and the manufacturer of the software on your computer.
- The CA's job is to digitally sign data strings containing the following information

  (Alice, Alice's public key).

  This data string, and the associated signature is called a digital certificate. The CA will only sign this data if it truly believes that the public key really does belong to Alice.
- When Alice now sends you her public key, contained in a digital certificate, you now trust that the purported key really is that of Alice, since you trust the CA to do its job correctly.

This use of a digital certificate binds the name 'Alice' with the 'Key', it is therefore often called an identity certificate. Other bindings are possible, we shall see some of these later related to authorizations.

Public key certificates will typically (although not always) be stored in repositories and accessed as required. For example, most browsers keep a list of the certificates that they have come across. The digital certificates do not need to be stored securely since they cannot be tampered with as they are digitally signed.

To see the advantage of certificates and CAs in more detail consider the following example of a world without a CA. In the following discussion we break with our colour convention for a moment and now use red to signal public keys which must be obtained in an authentic manner and blue to signal public keys which do not need to be obtained in an authentic manner.

In a world without a CA you obtain many individual public keys from each individual in some authentic fashion. For example

6A5DEF....A21	Jim Bean's public key,
7F341A....BFF	Jane Doe's public key,
B5F34A....E6D	Microsoft's update key.

Hence, each key needs to be obtained in an authentic manner, as does every new key you obtain.

Now consider the world with a CA. You obtain a single public key in an authentic manner, namely the CA's public key. We shall call our CA Ted since he is trustworthy. You then obtain many individual public keys, signed by the CA, in possibly an unauthentic manner. For example they could be attached at the bottom of an email, or picked up whilst browsing the web.

A45EFB....C45	Ted's totally trustworthy key,
6A5DEF....A21	Ted says 'This is Jim Bean's public key',
7F341A....BFF	Ted says 'This is Jane Doe's public key',
B5F34A....E6D	Ted says 'This is Microsoft's update key'.

If you trust Ted's key and you trust Ted to do his job correctly then you trust all the public keys you hold to be authentic.

In general a digital certificate is not just a signature on the single pair

(Alice, Alice's public key),

one can place all sorts of other, possibly application specific, information into the certificate. For example it is usual for the certificate to contain the following information.

- user's name,
- user's public key,
- is this an encryption or signing key?
- name of the CA,
- serial number of the certificate,
- expiry date of the certificate,
- ....

Commercial certificate authorities exist who will produce a digital certificate for your public key, often after payment of a fee and some checks on whether you are who you say you are. The certificates produced by commercial CAs are often made public, so one can call them public 'public key certificates', in that there use is mainly over open public networks.

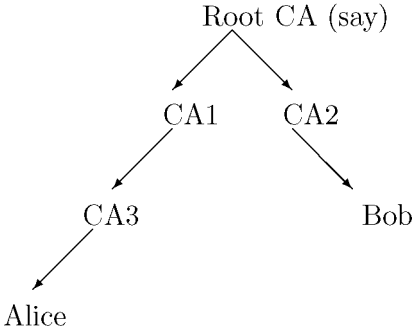
CAs are also used in proprietary systems, for example in debit/credit card systems or by large corporations. In such situations it may be the case that the end users do not want their public key certificates to be made public, in which case one can call them private 'public key certificates'. But one should bear in mind that whether the digital certificate is public or private should not effect the security of the private key associated to the public key contained in the certificate. The decision to make one's certificates private is often one of business rather than security.

It is common for more than one CA to exist. A quick examination of the properties of your web browser will reveal a large number of certificate authorities which your browser assumes you 'trust' to perform the function of a CA. As there are more than one CA it is common for one CA to sign a digital certificate containing the public key of another CA, and vice versa, a process which is known as cross-certification.

Cross-certification is needed if more than one CA exists, since a user may not have a trusted copy of the CA's public key needed to verify another user's digital certificate. This is solved by cross-certificates, i.e. one CA's public key is signed by another CA. The user first verifies the appropriate cross-certificate, and then verifies the user certificate itself.

With many CAs one can get quite long certificate chains, as Fig. 1 illustrates. Suppose Alice trusts the Root CA's public key and she obtains Bob's public key which is signed by the private key of CA2. She then obtains CA2's public key, either along with Bob's digital certificate or by some other means. CA2's public key comes in a certificate which is signed by the private key of the Root CA. Hence, by verifying all the signatures she ends up trusting Bob's public key.

FIGURE 1. Example certification hierarchy



Often the function of a CA is split into two parts. One part deals with verifying the user's identity and one part actually signs the public keys. The signing is performed by the CA, whilst the identity of the user is parcelled out to a registration authority, or RA. This can be a good practice, with the CA implemented in a more secure environment to protect the long-term private key.

The main problem with a CA system arises when a user's public key is compromised or becomes untrusted for some reason. For example

- a third party has gained knowledge of the private key,
- an employee leaves the company.

As the public key is no longer to be trusted all the associated digital certificates are now invalid and need to be revoked. But these certificates can be distributed over a large number of users, each one of which needs to be told to no longer trust this certificate. The CA must somehow inform all users that the certificate(s) containing this public key is/are no longer valid, in a process called certificate revocation.

One way to accomplish this is via a Certificate Revocation List, or CRL, which is a signed statement by the CA containing the serial numbers of all certificates which have been revoked by that CA and whose validity period has not expired. One clearly need not include in this the serial numbers of certificates which have passed their expiry date. Users must then ensure they have the latest CRL. This can be achieved by issuing CRLs at regular intervals even if the list has not changed. Such a system can work well in a corporate environment when overnight background jobs are often used to make sure each desktop computer in the company is up to date with the latest software etc. For other situations it is hard to see how the CRLs can be distributed, especially if there are a large number of CAs trusted by each user.

In summary, with secret key cryptography the main problems were ones of

- key management,
- key distribution.

These problems resulted in keys needing to be distributed via secure channels. In public key systems we replace these problems with those of

- key authentication,

in other words which key belongs to who. Hence, keys needed to be distributed via authentic channels. The use of digital certificates provides the authentic channels needed to distribute the public keys.

The whole system of CAs and certificates is often called the Public Key Infrastructure or PKI. This essentially allows a distribution of trust; the need to trust the authenticity of each individual public key in your possession is replaced by the need to trust a body, the CA, to do its job correctly.

In ensuring the CA does its job correctly you can either depend on the legal system, with maybe a state sponsored CA, or you can trust the business system in that it would not be in the CA's business interests to not act properly. For example, if it did sign a key in your name by mistake then you could apply publicly for exemplary restitution.

We end this section by noting that we have now completely solved the key distribution problem: For two users to agree on a shared secret key, they first obtain authentic public keys from a CA. Then secure session keys are obtained using, for example, signed Diffie–Hellman,

$$\begin{array}{ccc} \text{Alice} & & \text{Bob} \\ (g^a, \text{Sign}_{\text{Alice}}(g^a)) & \longrightarrow & \\ & \longleftarrow & (g^b, \text{Sign}_{\text{Bob}}(g^b)) \end{array}$$

3. Example Applications of PKI

In this section we shall look at some real systems which distribute trust via digital certificates. The examples will be

- PGP,
- SSL,
- X509 (or PKIX),
- SPKI.

3.1. PGP. The email encryption program Pretty Good Privacy, or PGP, takes a bottom-up approach to the distribution of trust. The design goals of PGP were to give a low-cost encryption/signature system for all users, hence the use of an expensive top-down global PKI would not fit this model. Instead the system makes use of what it calls a 'Web of Trust'.

The public key management is done from the bottom up by the users themselves. Each user acts as their own CA and signs other user's public keys. So Alice can sign Bob's public key and then Bob can give this signed 'certificate' to Charlie, in which case Alice is acting as a CA for Bob. If Charlie trusts Alice's judgement with respect to signing people's keys then she will trust that Bob's key really does belong to Bob. It is really up to Charlie to make this decision. As users keep doing this cross-certification of each other's keys, a web of trusted keys grows from the bottom up.

PGP itself as a program uses RSA public key encryption for low-volume data such as session keys. The block cipher used for bulk transmission is called IDEA. This block cipher has a 64-bit block size and a 128-bit key size and is used in CFB mode. Digital signatures in PGP can be produced either with the RSA or the DSA algorithm, after a message digest is taken using either MD5 or SHA-1.

The keys that an individual trusts are held in a so-called key ring. This means users have control over their own local public key store. This does not rule out a centralized public key store, but means one is not necessarily needed.

Key revocation is still a problem with PGP as with all such systems. The ad-hoc method adopted by PGP is that if your key is compromised then you should tell all your friends, who have a copy of your key, to delete the key from their key ring. All your friends should then tell all their friends and so on.

3.2. Secure Socket Layer. Whilst the design of PGP was driven by altruistic ideals, namely to provide encryption for the masses, the Secure Socket Layer, or SSL, was driven by commercial requirements, namely to provide a secure means for web based shopping and sales. Essentially SSL adds security to the TCP level of the IP stack. It provides security of data and not parties but allows various protocols to be transparently layered on top, for example HTTP, FTP, TELNET, etc.

The primary objective was to provide channel security, to enable the encrypted transmission of credit card details or passwords. After an initial handshake all subsequent traffic is encrypted. The server side of the communication, namely the website or the host computer in a Telnet session, is always authenticated for the benefit of the client. Optionally the client may be authenticated to the user, but this is rarely done in practice for web based transactions.

As in PGP, bulk encryption is performed using a block or stream cipher (usually either DES or an algorithm from the RC family). The choice of precise cipher is chosen between the client and server during the initial handshake. The session key to be used is derived using standard protocols such as the Diffie–Hellman protocol, or RSA based key transport.

The server is authenticated since it provides the client with an X509 public key certificate. This, for web shopping transactions, is signed by some global CA whose public key comes embedded into the user's web browser. For secure Telnet sessions (often named SSH after the program which runs them) the server side certificate is usually a self-signed certificate from the host computer.

The following is a simplified overview of how SSL can operate.

- The client establishes connection with the server on a special port number so as to signal this will be a secure session.
- The server sends a certified public key to the client.
- The client verifies the certificate and decides whether it trusts this public key.
- The client chooses a random secret.
- The client encodes this with the server's public key and sends this back to the server.
- The client and server now securely share the secret.
- The server now authenticates itself to the client by responding using the shared secret.

The determination of session keys can be a costly operation for both the server and the client, especially when the data may come in bursts, as when one is engaged in shopping transactions, or performing some remote access to a computer. Hence, there is some optimization made to enable reuse of session keys. The client is allowed to quote a previous session key, the server can either accept it or ask for a new one to be created. So as to avoid any problems this ability is limited by two rules. Firstly a session key should have a very limited lifetime and secondly any fatal error in any part of the protocols will immediately invalidate the session key and require a new one to be determined. In SSL the initial handshake is also used for the client and the server to agree on which bulk encryption algorithm to use, this is usually chosen from the list of RC4, RC5, DES or Triple DES.

3.3. X509 Certificates. When discussing SSL we mentioned that the server uses an X509 public key certificate. X509 is a standard which defines a structure for public key certificates, currently it is the most widely deployed certificate standard. A CA assigns a unique name to each

user and issues a signed certificate. The name is often the URL or email address. This can cause problems since, for example, many users may have different versions of the same email address. If you send a signed email containing your certificate for your email 'address'

N.P.Smart@some.where.com

but your email program sends this from the 'address'

Nigel.Smart@some.where.com

then, even though you consider both addresses to be equivalent, the email client of the recipient will often complain saying that the signature is not to be trusted.

The CAs are connected in a tree structure, with each CA issuing a digital certificate for the one beneath it. In addition cross-certification between the branches is allowed. The X509 certificates themselves are defined in standards using a language called ASN.1, or Abstract Syntax Notation. This can be rather complicated at first sight and the processing of all the possible options often ends up with incredible 'code bloat'.

The basic X509 certificate structure is very simple, but can end up being very complex in any reasonable application. This is because some advanced applications may want to add additional information into the certificates which enable authorization and other capabilities. However, the following records are always in a certificate.

- The version number of the X509 standard which this certificate conforms to.
- The certificate serial number.
- The CA's signing algorithm identifier. This should specify the algorithm and the domain parameters used, if the CA has multiple possible algorithms or domain parameters.
- The issuer's name, i.e. the name of the issuing CA.
- The validity period in the form of a not-before and not-after date.
- The subject's name, i.e. whose public key is being signed. This could be an email address or a domain name.
- The subject's public key. This contains the algorithm name and any associated domain parameters plus the actual value of the public key
- The issuer's signature on the subject's public key and all data that is to be bound to the subject's public key, such as the subject's name.

3.4. SPKI. In response to some of the problems associated with X509, another type of certificate format has been proposed called SPKI, or Simple Public Key Infrastructure. This system aims to bind authorizations as well as identities, and also tries to deal with the issue of delegation of authorizations and trust. Thus it may be suitable for business to business e-commerce transactions. For example, when managers go on holiday they can delegate their authorizations for certain tasks to their subordinates.

SPKI does not assume the global CA hierarchy which X509 does. It assumes a more ground-up approach like PGP. However, it is currently not used much commercially since PKI vendors have a lot of investment in X509 and are probably not willing to switch over to a new system (and the desktop applications such as your web browser would also need significant alterations).

Instead of using ASN.1 to describe certificates, SPKI uses S-expressions. These are LISP-like structures which are very simple to use and describe. In addition S-expressions can be made very simple even for humans to understand, as opposed to the machine-only readable formats of X509 certificates. S-expressions can even come with display hints to enable greater readability. The current draft standard specifies these display hints as simple MIME-types.

Each SPKI certificate has an issuer and a subject both of which are public keys (or a hash of a public key), and not names. This is because SPKI's authors claim that it is a key which does something and not a name. After all it is a key which is used to sign a document etc. Focusing on the keys also means we can concentrate more on the functionality. There are two types of

SPKI certificates: ones for binding identities to keys and ones for binding authorizations to keys. Internally these are represented as tuples of 4 and 5 objects, which we shall now explain.

3.4.1. *SPKI 4-Tuples.* To give an identity certificate and bind a name with a key, like X509 does, SPKI uses a 4-tuple structure. This is an internal abstraction of what the certificate represents and is given by:

$$\text{(Issuer, Name, Subject, Validity)}.$$

In real life this would consist of the following five fields:

- issuer's public key,
- name of the subject,
- subject's public key,
- validity period,
- signature of the issuer on the triple (Name, Subject, Validity).

Anyone is able to issue such a certificate, and hence become a CA.

3.4.2. *SPKI 5-Tuples.* 5-tuples are used to bind keys to authorizations. Again this is an internal abstraction of what the certificate represents and is given by

$$\text{(Issuer, Subject, Delegation, Authorization, Validity)}.$$

In real life this would consist of the following six fields:

- issuer's public key.
- subject's public key.
- delegation. A 'Yes' or 'No' flag, saying whether the subject can delegate the permission or not.
- authorization. What the subject is being given permission to do
- validity. How long the authorization is for.
- signature of the issuer on the quadruple (S,D,A,V).

One can combine an authorization certificate and an identity certificate to obtain an audit trail. This is needed since the authorization certificate only allows a key to perform an action. It does not say who owns the key. To find out who owns a key you need to use an identity certificate.

When certificate chains are eventually checked to enable some authorization, a 5-tuple reduction procedure is carried out. This can be represented by the following rule

$$\begin{aligned}(I_1, S_1, D_1, A_1, V_1) + (I_2, S_2, D_2, A_2, V_2)& \\ = (I_1, S_2, D_2, A_1 \cap A_2, V_1 \cap V_2)&.\end{aligned}$$

This equality holds only if

- $S_1 = I_2$
- $D_1 = \mathbf{true}$.

This means the first two certificates together can be interpreted as the third. This third 5-tuple is not really a certificate, it is the meaning of the first two when they are presented together.

As an example we will show how combining two 5-tuples is equivalent to delegating authority. Suppose our first 5-tuple is given by:

- I_1 = Alice
- S_1 = Bob
- $D_1 = \mathbf{true}$
- A_1 = Spend up to £100 on Alice's account
- V_1 = forever.

So Alice allows Bob to spend up to £100 on her account and allows Bob to delegate this authority to anyone he chooses.

Now consider the second 5-tuple given by

- I_2 = Bob
- S_2 = Charlie
- D_2 = **false**
- A_2 = Spend between £50 and £200 on Alice's account
- V_2 = before tomorrow morning.

So Bob is saying Charlie can spend between £50 and £200 of Alice's money, as long as it happens before tomorrow morning.

We combine these two 5-tuples, using the 5-tuple reduction rule, to form the new 5-tuple

- I_3 = Alice
- S_3 = Charlie
- D_3 = **false**
- A_3 = Spend between £50 and £100 on Alice's account
- V_3 = before tomorrow morning.

Since Alice has allowed Bob to delegate she has in effect allowed Charlie to spend between £50 and £100 on her account before tomorrow morning.

4. Other Applications of Trusted Third Parties

In some applications it is necessary for signatures to remain valid for a long time. Revocation of a public key, even long after the legitimate creation of the signature, potentially invalidates all digital signatures made using that key, even those in the past. This is a major problem if digital signatures are to be used for documents of long-term value such as wills, life insurance and mortgage contracts. We essentially need methods to prove that a digital signature was made prior to the revocation of the key and not after it. This brings us to the concept of time stamping.

A time stamping service is a means whereby a trusted entity will take a signed message, add a date/timestamp and sign the result using its own private key. This proves when a signature was made (like a notary service in standard life insurance). However, there is the requirement that the public key of the time stamping service must never be revoked. An alternative to the use of a time stamping service is the use of a secure archive for signed messages.

As another application of a trusted third-party consider the problem associated with keeping truly secret keys for encryption purposes.

- What if someone loses or forgets a key? They could lose all your encrypted data.
- What if the holder of the key resigns from the company or is killed? The company may now want access to the encrypted data.
- What if the user is a criminal? Here the government may want access to the encrypted data.

One solution is to deposit a copy of your key with someone else in case you lose yours, or something untoward happens. On the other hand, simply divulging the key to anybody, even the government, is very insecure.

A proposed solution is key escrow implemented via a secret sharing scheme. Here the private key is broken into pieces, each of which can be verified to be correct. Each piece is then given to some authority. At some later point if the key needs to be recovered then a subset of the authorities can come together and reconstruct it from their shares. The authorities implementing this escrow facility are another example of a Trusted Third Party, since you really have to trust them. In fact the trust required is so high that this solution has been a source of major debate within the cryptographic and governmental communities in the past. The splitting of the key between the various escrow authorities can be accomplished using the trick of a secret sharing scheme which we will discuss in Chapter 23.

5. Implicit Certificates

One issue with digital certificates is that they can be rather large. Each certificate needs to at least contain both the public key of the user and the signature of the certificate authority on that key. This can lead to quite large certificate sizes, as the following table demonstrates:

	RSA	DSA	EC-DSA
User's key	1024	1024	160
CA sig	1024	320	320

This assumes for RSA keys one uses a 1024-bit modulus, for DSA one uses a 1024-bit prime p and a 160-bit prime q and for EC-DSA one uses a 160-bit curve. Hence, for example, if the CA is using 1024-bit RSA and they are signing the public key of a user using 1024-bit DSA then the total certificate size must be at least 2048 bits. An interesting question is whether this can be made smaller.

Implicit certificates enable this. An implicit certificate looks like

$$X|Y$$

where

- X is the data being bound to the public key,
- Y is the implicit certificate on X.

From Y we need to be able to recover the public key being bound to X and implicit assurance that the certificate was issued by the CA. In the system we describe below, based on a DSA or EC-DSA, the size of Y will be 1024 or 160 bits respectively. Hence, the size of the certificate is reduced to the size of the public key being certified.

5.1. System Setup. The CA chooses a public group G of known order n and an element $P \in G$. The CA then chooses a long-term private key c and computes the public key

$$Q = P^c.$$

This public key should be known to all users.

5.2. Certificate Request. Suppose Alice wishes to request a certificate and the public key associated to the information ID, which could be her name. Alice computes an ephemeral secret key t and an ephemeral public key

$$R = P^t.$$

Alice sends R and ID to the CA.

5.3. Processing of the request. The CA checks that he wants to link ID with Alice. The CA picks another random number k and computes

$$g = P^k R = P^k P^t = P^{k+t}.$$

Then the CA computes

$$s = cH(ID\|g) + k \pmod{n}.$$

Then the CA sends back to Alice the pair

$$(g, s).$$

The implicit certificate is the pair

$$(ID, g).$$

We now have to convince you that

- Alice can recover a valid public/private key pair,
- any other user can recover Alice's public key from this implicit certificate

5.4. Alice's Key Discovery. Alice knows the following information

$$t, s, R = P^t.$$

From this she can recover her private key

$$a = t + s \pmod{n}.$$

Note, Alice's private key is known only to Alice and not to the CA. In addition Alice has contributed some randomness t to her private key, as has the CA who contributed k. Her public key is then

$$P^a = P^{t+s} = P^t P^s = R \cdot P^s.$$

5.5. User's Key Discovery. Since s and R are public, a user, say Bob, can recover Alice's public key from the above message flows via

$$R \cdot P^s.$$

But this says nothing about the linkage between the CA, Alice's public key and the ID information. Instead Bob recovers the public key from the implicit certificate

$$(ID, g)$$

and the CA's public key

$$Q$$

via the equation

$$P^a = Q^{H(ID \| g)} g.$$

As soon as Bob sees Alice's key used in action, say he verifies a signature purported to have been made by Alice, he knows implicitly that it must have been issued by the CA, since otherwise Alice's signature would not verify correctly.

There are a number of problems with the above system which mean that implicit certificates are not used much in real life. For example,

(1) What do you do if the CA's key is compromised? Usually you pick a new CA key and re-certify the user's keys. But you cannot do this since the user's public key is chosen interactively during the certification process.
(2) Implicit certificates require the CA and users to work at the same security level. This is not considered good practice, as usually one expects the CA to work at a higher security level (say 2048-bit DSA) than the user (say 1024-bit DSA).

However for devices with restricted bandwidth they can offer a suitable alternative where traditional certificates are not available.

6. Identity Based Cryptography

Another way of providing authentic public keys, without the need for certificates, is to use a system whereby the user's key is given by their identity. Such a system is called an identity based encryption scheme or an identity based signature scheme. Such systems do not remove the need for a trusted third-party to perform the original authentication of the user, but they do however remove the need for storage and transmission of certificates.

The first scheme of this type was a signature scheme invented by Shamir in 1984. It was not until 2001 however that an identity based encryption scheme was given by Boneh and Franklin. We shall only describe the original identity based signature scheme of Shamir, which is based on the RSA problem.

A trusted third-party first calculates an RSA modulus N, keeping the two factors secret. The TTP also publishes a public exponent e, keeping the corresponding private exponent d to themselves. In addition there is decided a mapping

$$I : \{0,1\}^* \longrightarrow (\mathbb{Z}/N\mathbb{Z})^*$$

which takes bit strings to elements of $(\mathbb{Z}/N\mathbb{Z})^*$. Such a mapping could be implemented by a hash function.

Now suppose Alice wishes to obtain the private key g corresponding to her name 'Alice'. This is calculated for her by the TTP using the equation

$$g = I(\text{Alice})^d \pmod{N}.$$

To sign a message m, Alice generates the pair (t, s) via the equations

$$t = r^e \pmod{N},$$
$$s = g \cdot r^{H(m\|t)} \pmod{N},$$

where r is a random integer and H is a hash function.

To verify the signature (t, s) on the message m another user can do this, simply by knowing the TTP's public data and the identity of Alice, by checking that the following equation holds modulo N,

$$\begin{aligned} I(\text{Alice}) \cdot t^{H(m\|t)} &= g^e \cdot r^{e \cdot H(m\|t)} \\ &= \left(g \cdot r^{H(m\|t)}\right)^e \\ &= s^e. \end{aligned}$$

Chapter Summary

- Digital certificates allow us to bind a public key to some other information, such as an identity.
- This binding of key with identity allows us to solve the problem of how to distribute authentic public keys.
- Various PKI systems have been proposed, all of which have problems and benefits associated with them.
- PGP and SPKI work from the bottom up, whilst X509 works in a top-down manner.
- SPKI contains the ability to delegate authorizations from one key to another.
- Other types of trusted third-party applications exist such as time stamping and key escrow.
- Implicit certificates aim to reduce the bandwidth requirements of standard certificates, however they come with a number of drawbacks.
- Identity based cryptography helps authenticate a user's public key by using their identity as the public key, but it does not remove the need for trusted third parties.

Further Reading

A good overview of the issues related to PKI can be found in the book by Adams and Lloyd. For further information on PGP and SSL look at the books by Garfinkel and Rescorla.

C. Adams and S. Lloyd. *Understanding Public-Key Infrastructure: Concepts, Standards and Deployment Considerations.* New Riders Publishing, 1999.

S. Garfinkel. *PGP: Pretty Good Privacy.* O'Reilly & Associates, 1994.

E. Rescorla. *SSL and TLS: Design and Building Secure Systems.* Addison-Wesley, 2000.

Part 4

Security Issues

Having developed the basic public key cryptographic primitives we require, we now show that this is not enough. Often weaknesses occur because the designers of a primitive do not envisage how the primitive is actually going to be used in real life. We first outline a number of attacks, and then we try and define what it means to be secure; this leads us to deduce that the primitives we have given earlier are not really secure enough.

This then leads us to look at two approaches to build secure systems. One based on pure complexity theory ends up being doomed to failure, since pure complexity theory is about worst case rather than average case hardness of problems. The second approach of provable security ends up being more suitable and has in fact turned out to be highly influential on modern cryptography. This second approach also derives from complexity theory, but is based on the relative average case hardness of problems, rather than the absolute hardness of the worst case instance.

CHAPTER 17

Attacks on Public Key Schemes

Chapter Goals

- To explain Wiener's attack based on continued fractions.
- To describe lattice basis reduction algorithms and give some examples of how they are used to break cryptographic systems.
- To explain the technique of Coppersmith for finding small roots of modular polynomial equations and describe some of the cryptographic applications.
- To introduce the notions of partial key exposure and fault analysis.

1. Introduction

In this chapter we explain a number of attacks against naive implementations of schemes such as RSA and DSA. We shall pay particular attention to the techniques of Coppersmith, based on lattice basis reduction. What this chapter aims to do is show you that even though a cryptographic primitive such as the RSA function

$$x \longrightarrow x^e \pmod{N}$$

is a trapdoor one-way permutation, this on its own is not enough to build secure encryption systems. It all depends on how you use the RSA function. In later chapters we go on to show how one can build secure systems out of the RSA function and the other public key primitives we have met.

2. Wiener's Attack on RSA

We have mentioned in earlier chapters that often one uses a small public RSA exponent e so as to speed up the public key operations in RSA. Sometimes we have applications where it is more important to have a fast private key operation. Hence, one could be tempted to choose a small value of the private exponent d. Clearly this will lead to a large value of the encryption exponent e and we cannot choose too small a value for d, otherwise an attacker could find d using exhaustive search. However, it turns out that d needs to be at least the size of $\frac{1}{3}N^{1/4}$ due to an ingenious attack by Wiener which uses continued fractions.

Let $\alpha \in \mathbb{R}$, we define the following sequences, starting with $\alpha_0 = \alpha$, $p_0 = a_0$ and $q_0 = 1$, $p_1 = a_0a_1 + 1$ and $q_1 = a_1$,

$$\begin{aligned} a_i &= \lfloor \alpha_i \rfloor, \\ \alpha_{i+1} &= \frac{1}{\alpha_i - a_i}, \\ p_i &= a_i p_{i-1} + p_{i-2} \text{ for } i \geq 2, \\ q_i &= a_i q_{i-1} + q_{i-2} \text{ for } i \geq 2. \end{aligned}$$

The integers $a_0, a_1, a_2, \ldots$ are called the continued fraction expansion of α and the fractions

$$\frac{p_i}{q_i}$$

are called the convergents. The denominators of these convergents grow at an exponential rate and the convergent above is a fraction in its lowest terms since one can show

$$\gcd(p_i, q_i) = 1$$

for all values of i.

The important result is that if p and q are two integers with

$$\left|\alpha - \frac{p}{q}\right| \leq \frac{1}{2q^2}$$

then $\frac{p}{q}$ is a convergent in the continued fraction expansion of α.

Wiener's attack uses continued fractions as follows. We assume we have an RSA modulus $N = pq$ with $q < p < 2q$. In addition assume that the attacker knows that we have a small decryption exponent $d < \frac{1}{3}N^{1/4}$. The encryption exponent e is given to the attacker, where this exponent satisfies

$$ed = 1 \pmod{\Phi},$$

with

$$\Phi = \phi(N) = (p-1)(q-1).$$

We also assume $e < \Phi$, since this holds in most systems. First notice that this means there is an integer k such that

$$ed - k\Phi = 1.$$

Hence, we have

$$\left|\frac{e}{\Phi} - \frac{k}{d}\right| = \frac{1}{d\Phi}.$$

Now, $\Phi \approx N$, since

$$|N - \Phi| = |p + q - 1| < 3\sqrt{N}.$$

So we should have that $\frac{e}{N}$ is a close approximation to $\frac{k}{d}$.

$$\begin{aligned}
\left|\frac{e}{N} - \frac{k}{d}\right| &= \left|\frac{ed - Nk}{dN}\right| \\
&= \left|\frac{ed - k\Phi - Nk + k\Phi}{dN}\right| \\
&= \left|\frac{1 - k(N - \Phi))}{dN}\right| \\
&\leq \left|\frac{3k\sqrt{N}}{dN}\right| \\
&= \frac{3k}{d\sqrt{N}}.
\end{aligned}$$

Since $e < \Phi$, it is clear that we have $k < d$, which is itself less than $\frac{1}{3}N^{1/4}$ by assumption. Hence,

$$\left|\frac{e}{N} - \frac{k}{d}\right| < \frac{1}{2d^2}.$$

Since $\gcd(k, d) = 1$ we see that $\frac{k}{d}$ will be a fraction in its lowest terms. Hence, the fraction

$$\frac{k}{d}$$

must arise as one of the convergents of the continued fraction expansion of

$$\frac{e}{N}.$$

The correct one can be detected by simply testing which one gives a d which satisfies

$$(m^e)^d = m \pmod{N}$$

for some random value of m. The total number of convergents we will need to take is of order $O(\log N)$, hence the above gives a linear time algorithm to determine the private exponent when it is less than $\frac{1}{3}N^{1/4}$.

As an example suppose we have the RSA modulus

$$N = 9\,449\,868\,410\,449$$

with the public key

$$e = 6\,792\,605\,526\,025.$$

We are told that the decryption exponent satisfies $d < \frac{1}{3}N^{1/4} \approx 584$. To apply Wiener's attack we compute the continued fraction expansion of the number

$$\alpha = \frac{e}{N},$$

and check each denominator of a convergent to see whether it is equal to the private key d. The convergents of the continued fraction expansion of α are given by

$$1, \frac{2}{3}, \frac{3}{4}, \frac{5}{7}, \frac{18}{25}, \frac{23}{32}, \frac{409}{569}, \frac{1659}{2308}, \ldots$$

Checking each denominator in turn we see that the decryption exponent is given by

$$d = 569,$$

which is the denominator of the 7th convergent.

3. Lattices and Lattice Reduction

We shall see later in this chapter that lattice basis reduction provides a crucial tool in various attacks on a number of public key systems. So we spend this section giving an overview of this important area. We first need to recap on some basic linear algebra.

Suppose $x = (x_1, x_2, \ldots, x_n)$ is an n-dimension real vector, i.e. for all i we have $x_i \in \mathbb{R}$. The set of all such vectors is denoted $\mathbb{R}^n$. On two such vectors we can define an *inner product*

$$\langle x, y \rangle = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n,$$

which is a function from pairs of n-dimensional vectors to the real numbers. You probably learnt at school that two vectors x and y are orthogonal, or meet at right angles, if and only if we have

$$\langle x, y \rangle = 0.$$

Given the inner product we can then define the size, or length, of a vector by

$$\|x\| = \sqrt{\langle x, x \rangle} = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}.$$

This length corresponds to the intuitive notion of length of vectors which we have, in particular the length satisfies a number of properties.

- $\|x\| \geq 0$, with equality if and only if x is the zero vector.

- Triangle inequality: For two n-dimensional vectors x and y
$$\|x+y\| \leq \|x\| + \|y\|.$$
- Scaling: For a vector x and a real number a
$$\|ax\| = |a| \cdot \|x\|.$$

A set of vectors $\{b_1, \ldots, b_m\}$ in $\mathbb{R}^n$ is called linearly independent if the equation
$$a_1 b_1 + \cdots + a_m b_m = 0,$$
for real numbers a_i, implies that all a_i are equal to zero. If the set is linearly independent then we must have $m \leq n$.

Suppose we have a set of m linearly independent vectors, $\{b_1, \ldots, b_m\}$. We can look at the set of all real linear combinations of these vectors,
$$V = \left\{ \sum_{i=1}^{m} a_i b_i : a_i \in \mathbb{R} \right\}.$$
This is a vector subspace of $\mathbb{R}^n$ of dimension m and the set $\{b_1, \ldots, b_m\}$ is called a basis of this subspace. If we form the matrix B consisting of the ith column of B being equal to b_i then we have
$$V = \{Ba : a \in \mathbb{R}^m\}.$$
The matrix B is called the basis matrix.

Every subspace V has a large number of possible basis matrices. Given one such basis it is often required to produce a basis with certain prescribed nice properties. Often in applications throughout science and engineering one requires a basis which is pairwise orthogonal, i.e.
$$\langle b_i, b_j \rangle = 0$$
for all $i \neq j$. Luckily there is a well-known method which takes one basis, $\{b_1, \ldots, b_m\}$ and produces a basis $\{b_1^*, \ldots, b_m^*\}$ which is pairwise orthogonal. This method is called the Gram–Schmidt process and the basis $\{b_1^*, \ldots, b_m^*\}$ produced from $\{b_1, \ldots, b_m\}$ via this process is called the Gram–Schmidt basis corresponding to $\{b_1, \ldots, b_m\}$. One computes the b_i^* from the b_i via the equations
$$\mu_{i,j} = \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle}, \text{ for } 1 \leq j < i \leq n,$$
$$b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^*.$$

For example if we have
$$b_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix} \text{ and } b_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$
then we compute
$$b_1^* = b_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix},$$
$$b_2^* = b_2 - \mu_{2,1} b_1^* = \begin{pmatrix} 1 \\ 1 \end{pmatrix} - \frac{1}{2} \begin{pmatrix} 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$
since
$$\mu_{2,1} = \frac{\langle b_2, b_1^* \rangle}{\langle b_1^*, b_1^* \rangle} = \frac{2}{4} = \frac{1}{2}.$$
Notice how we have $\langle b_1^*, b_2^* \rangle = 0$, so the new Gram–Schmidt basis is orthogonal.

A *lattice* is like the vector subspace V above, but given a set of basis vectors $\{b_1, \ldots, b_m\}$ instead of taking the real linear combinations of the b_i we are only allowed to take the *integer* linear combinations of the b_i,

$$L = \left\{ \sum_{i=1}^{m} a_i b_i : a_i \in \mathbb{Z} \right\} = \{Ba : a \in \mathbb{Z}^m\}.$$

The set $\{b_1, \ldots, b_m\}$ is still called the set of basis vectors and the matrix B is still called the basis matrix. To see why lattices are called lattices, consider the lattice L generated by the two vectors

$$b_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix} \text{ and } b_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

This is the set of all vectors of the form

$$\begin{pmatrix} 2x + y \\ y \end{pmatrix},$$

where $x, y \in \mathbb{Z}$. If one plots these points in the plane then one sees that these points form a 2-dimensional lattice.

A lattice is a discrete version of a vector subspace. Since it is discrete there is a well-defined smallest element, bar the trivially small element of the zero vector of course. Many hard tasks in computing, and especially cryptography, can be reduced to trying to determine the smallest non-zero vector in a lattice. We shall see some of these applications later, but before continuing with our discussion on lattices in general we pause to note that it is generally considered to be a hard problem to determine the smallest non-zero vector in an arbitrary lattice. Later we shall see that whilst this problem is hard in general it is in fact easy in low dimension, a situation which we shall use to our advantage later on.

Let us now return to considering lattices in general. Just as with vector subspaces, given a lattice basis one could ask is there a nicer basis? Suppose B is a basis matrix for a lattice L. To obtain another basis matrix B' the only operation we are allowed to do is post-multiply B by a uni-modular integer matrix. This means we must have

$$B' = BU$$

for some integer matrix U with $\det(U) = \pm 1$. This means that the absolute value of the determinant of a basis matrix of a lattice is an invariant of the lattice, i.e. it does not depend on the choice of basis. Given a basis matrix B we call

$$\Delta = |\det(B^t B)|^{1/2}$$

the *discriminant* of the lattice. If L is a lattice of full rank, i.e. B is a square matrix, then we have

$$\Delta = |\det(B)|.$$

One could ask, given a lattice L does there exist an orthogonal basis? In general the answer to this last question is no. If one looks at the Gram–Schmidt process in more detail one sees that, even if one starts out with integer vectors, the coefficients $\mu_{i,j}$ almost always end up not being integers. Hence, whilst the Gram–Schmidt basis vectors span the same vector subspace as the original basis they do not span the same lattice as the original basis. This is because we are not allowed to make a change of basis which consists of non-integer coefficients. However, we could try and make a change of basis so that the new basis is 'close' to being orthogonal in that

$$|\mu_{i,j}| \leq \frac{1}{2} \text{ for } 1 \leq j < i \leq n.$$

These considerations led Lenstra, Lenstra and Lovász to define the following notion of reduced basis, called an LLL reduced basis after its inventors:

DEFINITION 17.1. *A basis* $\{b_1, \ldots, b_m\}$ *is called LLL reduced if the associated Gram–Schmidt basis* $\{b_1^*, \ldots, b_m^*\}$ *satisfies*

$$|\mu_{i,j}| \leq \frac{1}{2} \text{ for } 1 \leq j < i \leq m, \tag{13}$$

$$\|b_i^*\|^2 \geq \left(\frac{3}{4} - \mu_{i,i-1}^2\right) \|b_{i-1}^*\|^2 \text{ for } 1 < i \leq m. \tag{14}$$

What is truly amazing about an LLL reduced basis is

- An LLL reduced basis can be computed in polynomial time, see below for the method.
- The first vector in the reduced basis is very short, in fact it is close to the shortest non-zero vector in that for all non-zero $x \in L$ we have
$$\|b_1\| \leq 2^{(m-1)/2}\|x\|.$$
- $\|b_1\| \leq 2^{m/4}\Delta^{1/m}$.

The constant $2^{(m-1)/2}$ in the second bullet point above is a worst case constant. In practice for many lattices of reasonable dimension after one applies the LLL algorithm to obtain an LLL reduced basis, the first vector in the LLL reduced basis is in fact equal to the smallest vector in the lattice.

The LLL algorithm works as follows: We keep track of a copy of the current lattice basis B and the associated Gram–Schmidt basis B^*. At any point in time we are examining a fixed column k, where we start with $k = 2$.

- If condition (13) does not hold for $\mu_{k,j}$ with $1 \leq j < k$ then we alter the basis B so that it does.
- If condition (14) does not hold for column k and column $k-1$ we swap columns k and $k-1$ around and decrease the value of k by one (unless k is already equal to two). If condition (14) holds then we increase k by one.

At some point we will obtain $k = m$ and the algorithm terminates. In fact one can show that the number of iterations where one decreases k is bounded and so one is guaranteed for the algorithm to terminate. When the algorithm terminates it clearly will produce an LLL reduced basis.

As an example we take the above example basis of a two-dimensional lattice in $\mathbb{R}^2$, we take the lattice basis

$$b_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \; b_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

The associated Gram–Schmidt basis is given by

$$b_1^* = \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \; b_2^* = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

But this is not a basis of the associated lattice since one cannot pass from $\{b_1, b_2\}$ to $\{b_1^*, b_2^*\}$ via a unimodular integer transformation.

We now apply the LLL algorithm with $k = 2$ and find that the first condition (13) is satisfied since $\mu_{2,1} = \frac{1}{2}$. However, the second condition (14) is not satisfied because

$$1 = \|b_2^*\|^2 \leq \left(\frac{3}{4} - \mu_{2,1}^2\right) \|b_1^*\|^2 = \frac{1}{2} \cdot 4 = 2.$$

Hence, we need to swap the two basis vectors around and compute the corresponding Gram–Schmidt vectors. We then obtain the new lattice basis vectors

$$b_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \; b_2 = \begin{pmatrix} 2 \\ 0 \end{pmatrix},$$

with the associated Gram–Schmidt basis vectors

$$b_1^* = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \; b_2^* = \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

We now go back to the first condition again. This time we see that we have $\mu_{2,1} = 1$ which violates the first condition. To correct this we subtract one lot of b_1 from the vector b_2 so as to obtain $\mu_{2,1} = 0$. We now find the lattice basis is given by

$$b_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \; b_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

and the Gram–Schmidt basis is then identical to this lattice basis, in that

$$b_1^* = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \; b_2^* = \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

Now we are left to check the second condition again, we find

$$2 = \|b_2^*\|^2 \geq \left(\frac{3}{4} - \mu_{2,1}^2\right) \|b_1^*\|^2 = \frac{3}{4} \cdot 2 = \frac{3}{2}.$$

Hence, both conditions are satisfied and we conclude that

$$b_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \; b_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

is a reduced basis for the lattice L.

We end this introduction to lattices by noticing that there is a link between continued fractions and lattice reduction. Applying the continued fraction algorithm to a real number α is used to find values of p and q, with q not too large, so that

$$|q\alpha - p|$$

is small. A similar effect can be achieved using lattices by applying the LLL algorithm to the lattice generated by the columns of the matrix

$$\begin{pmatrix} 1 & 0 \\ C\alpha & -C \end{pmatrix},$$

for some constant C. This is because the lattice L contains the 'short' vector

$$\begin{pmatrix} q \\ C(q\alpha - p) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ C\alpha & -C \end{pmatrix} \cdot \begin{pmatrix} q \\ p \end{pmatrix}.$$

In some sense we therefore can consider the LLL algorithm to be a multi-dimensional generalization of the continued fraction algorithm.

4. Lattice Based Attacks on RSA

In this section we examine how lattices can be used to attack certain systems, when some other side information is known. This is analogous to Wiener's attack above where one could break RSA given the side information that the decryption exponent was exceedingly short. Much of the work in this area is derived from initial work of Coppersmith, which was later simplified by Howgrave-Graham.

The basic technique is a method to solve the following problem: Given a polynomial

$$f(x) = f_0 + f_1 x + \cdots + f_{d-1} x^{d-1} + x^d$$

over the integers of degree d and the side information that there exists a root x_0 modulo N which is small, say $|x_0| < N^{1/d}$, can one efficiently find the small root x_0? The answer is surprisingly yes, and this leads to a number of interesting cryptographic consequences.

The basic idea is to find a polynomial $h(x) \in \mathbb{Z}[x]$ which has the same root modulo N as the target polynomial $f(x)$ does. This new polynomial $h(x)$ should be small in the sense that the norm of its coefficients,

$$\|h\|^2 = \sum_{i=0}^{\deg(h)} h_i^2$$

should be small. If such an $h(x)$ can be found then we can appeal to the following lemma.

LEMMA 17.2. *Let $h(x) \in \mathbb{Z}[x]$ denote a polynomial of degree at most n and let X and N be positive integers. Suppose*

$$\|h(xX)\| < N/\sqrt{n}$$

then if $|x_0| < X$ satisfies

$$h(x_0) = 0 \pmod{N}$$

then $h(x_0) = 0$ over the integers and not just modulo N.

Now we return to our original polynomial $f(x)$ of degree d and notice that if

$$f(x_0) = 0 \pmod{N}$$

then we also have

$$f(x_0)^k = 0 \pmod{N^k}.$$

Moreover, if we set, for some given value of m,

$$g_{u,v}(x) = N^{m-v} x^u f(x)^v$$

then

$$g_{u,v}(x_0) = 0 \pmod{N^m}$$

for all $0 \le u < d$ and $0 \le v \le m$. We then fix m and try to find $a_{u,v} \in \mathbb{Z}$ so that

$$h(x) = \sum_{u \ge 0} \sum_{v=0}^{m} a_{u,v} g_{u,v}(x)$$

satisfies the conditions of the above lemma.

In other words we wish to find integer values of $a_{u,v}$ so that the resulting polynomial h satisfies

$$\|h(xX)\| \le N^m/\sqrt{d(m+1)},$$

with

$$h(xX) = \sum_{u \ge 0} \sum_{v=0}^{m} a_{u,v} g_{u,v}(xX).$$

This is a minimization problem which can be solved using lattice basis reduction, as we shall now show in a simple example.

Suppose our polynomial $f(x)$ is given by

$$f(x) = x^2 + ax + b$$

and we wish to find an x_0 such that

$$f(x_0) = 0 \pmod{N}.$$

We set $m = 2$ in the above construction and compute

$$\begin{aligned}
g_{0,0}(xX) &= N^2,\\
g_{1,0}(xX) &= XN^2x,\\
g_{0,1}(xX) &= bN + aXNx + NX^2x^2,\\
g_{1,1}(xX) &= bNXx + aNX^2x^2 + NX^3x^3,\\
g_{0,2}(xX) &= b^2 + 2baXx + (a^2+2b)X^2x^2 + 2aX^3x^3 + X^4x^4,\\
g_{1,2}(xX) &= b^2Xx + 2baX^2x^2 + (a^2+2b)X^3x^3 + 2aX^4x^4 + X^5x^5.
\end{aligned}$$

We are asking for a linear combination of the above six polynomials such that the resulting polynomial has small coefficients. Hence we are led to look for small vectors in the lattice generated by the columns of the following matrix, where each column represents one of the polynomials above and each row represents a power of x,

$$A = \begin{pmatrix}
N^2 & 0 & bN & 0 & b^2 & 0\\
0 & XN^2 & aXN & bNX & 2abX & Xb^2\\
0 & 0 & NX^2 & aNX^2 & (a^2+2b)X^2 & 2abX^2\\
0 & 0 & 0 & NX^3 & 2aX^3 & (a^2+2b)X^3\\
0 & 0 & 0 & 0 & X^4 & 2aX^4\\
0 & 0 & 0 & 0 & 0 & X^5
\end{pmatrix}.$$

This matrix has determinant equal to

$$\det(A) = N^6X^{15},$$

and so applying the LLL algorithm to this matrix we obtain a new lattice basis B. The first vector b_1 in B will satisfy

$$\|b_1\| \leq 2^{6/4}\det(A)^{1/6} = 2^{3/2}NX^{5/2}.$$

So if we set $b_1 = Au$, with $Bu = (u_1, u_2, \ldots, u_6)^t$, then we form the polynomial

$$h(x) = u_1g_{0,0}(x) + u_2g_{1,0}(x) + \cdots + u_6g_{1,2}(x)$$

then we will have

$$\|h(xX)\| \leq 2^{3/2}NX^{5/2}.$$

To apply Lemma 17.2 we will require that

$$2^{3/2}NX^{5/2} < N^2/\sqrt{6}.$$

Hence by determining an integer root of $h(x)$ we will determine the small root x_0 of $f(x)$ modulo N assuming that

$$|x_0| \leq X = \frac{N^{2/5}}{48^{1/5}}.$$

In particular this will work when $|x_0| \leq N^{0.39}$.

A similar technique can be applied to any polynomial of degree d so as to obtain

THEOREM 17.3 (Coppersmith). *Let $f \in \mathbb{Z}[x]$ be a monic polynomial of degree d and N an integer. If there is some root x_0 of f modulo N such that $|x_0| \leq X = N^{1/d-\epsilon}$ then one can find x_0 in time a polynomial in $\log N$ and $1/\epsilon$, for fixed values of d.*

Similar considerations apply to polynomials in two variables, the analogue of Lemma 17.2 becomes:

LEMMA 17.4. *Let $h(x,y) \in \mathbb{Z}[x,y]$ denote a sum of at most w monomials and suppose*

- $h(x_0, y_0) = 0 \pmod{N^e}$ *for some positive integers* N *and* e *where the integers* x_0 *and* y_0 *satisfy* $|x_0| < X$ *and* $|y_0| < Y$,
- $\|h(xX, yY)\| < N^e/\sqrt{w}$.

Then $h(x_0, y_0) = 0$ *holds over the integers.*

However, the analogue of Theorem 17.3 then becomes only a heuristic result.

We now use Theorem 17.3 and its generalizations to describe a number of attacks against RSA.

4.1. Hastad's Attack. In Chapter 11 we saw the following attack on the RSA system: Given three public keys (N_i, e_i) all with the same encryption exponent $e_i = 3$, if a user sent the same message to all three public keys then an adversary could recover the plaintext using the Chinese Remainder Theorem.

Now suppose that we protect against this attack by insisting that before encrypting a message m we first pad with some user-specific data. For example the ciphertext becomes, for user i,

$$c_i = (i \cdot 2^h + m)^3 \pmod{N_i}.$$

However, one can still break this system using an attack due to Hastad. Hastad's attack is related to Coppersmith's Theorem since we can interpret the attack scenario, generalized to k users and public encryption exponent e, as being given k polynomials of degree e

$$g_i(x) = (i \cdot 2^h + x)^e - c_i,\ 1 \le i \le k.$$

Then given that there is an m such that

$$g_i(m) = 0 \pmod{N_i},$$

the goal is to recover m. We can assume that m is smaller than any one of the moduli N_i. Setting

$$N = N_1 N_2 \cdots N_k$$

and using the Chinese Remainder Theorem we can find t_i so that

$$g(x) = \sum_{i=1}^{k} t_i g_i(x)$$

and

$$g(m) = 0 \pmod{N}.$$

Then, since g has degree e and is monic, using Theorem 17.3 we can recover m in polynomial time, as long as we have at least as many ciphertexts as the encryption exponent i.e. $k > e$, since

$$m < \min_i N_i < N^{1/k} < N^{1/e}.$$

4.2. Franklin–Reiter Attack and Coppersmith's Generalization. Now suppose we have one RSA public key (N, e) owned by Alice. The Franklin–Reiter attack applies to the following situation: Bob wishes to send two related messages m_1 and m_2 to Alice, where the relation is given by the public polynomial

$$m_1 = f(m_2) \pmod{N}.$$

Given c_1 and c_2 an attacker has a good chance of determining m_1 and m_2 for any small encryption exponent e. The attack is particularly simple when

$$f = ax + b \text{ and } e = 3,$$

with a and b fixed and given to the attacker. The attacker knows that m_2 is a root, modulo N, of the two polynomials

$$g_1(x) = x^3 - c_2,$$
$$g_2(x) = f(x)^3 - c_1.$$

So the linear factor $x - m_2$ divides both $g_1(x)$ and $g_2(x)$.

We now form the greatest common divisor of $g_1(x)$ and $g_2(x)$. Strictly speaking this is not possible in general since $\mathbb{Z}/N\mathbb{Z}[x]$ is not a Euclidean ring, but if the Euclidean algorithm breaks down then we would find a factor of N and so be able to find Alice's private key in any case. One can show that when

$$f = ax + b \text{ and } e = 3,$$

the resulting gcd, when it exists, must always be the linear factor $x - m_2$, and so the attacker can always find m_2 and then m_1.

Coppersmith extended the attack of Franklin and Reiter, in a way which also extends the padding result from Hastad's attack. Suppose before sending a message m we pad it with some random data. So for example if N is an n-bit RSA modulus and m is a k-bit message then we could append $n - k$ random bits to either the top or bottom of the message. Say

$$m' = 2^{n-k}m + r$$

where r is some, per message, random number of length $n - k$. Coppersmith showed that this padding is insecure.

Suppose Bob sends the same message to Alice twice, i.e. we have ciphertexts c_1 and c_2 corresponding to the messages

$$m_1 = 2^{n-k}m + r_1,$$
$$m_2 = 2^{n-k}m + r_2,$$

where r_1, r_2 are two different random $(n - k)$-bit numbers. The attacker sets $y_0 = r_2 - r_1$ and is led to solve the simultaneous equations

$$g_1(x, y) = x^e - c_1,$$
$$g_2(x, y) = (x + y)^e - c_2.$$

The attacker forms the resultant $h(y)$ of $g_1(x, y)$ and $g_2(x, y)$ with respect to x. Now $y_0 = r_2 - r_1$ is a small root of the polynomial $h(y)$, which has degree e^2. Using Coppersmith's Theorem 17.3 the attacker recovers $r_2 - r_1$ and then recovers m_2 using the method of the Franklin–Reiter attack.

Whilst the above trivial padding scheme is therefore insecure, one can find secure padding schemes for the RSA encryption algorithm. We shall return to padding schemes for RSA in Chapter 20.

4.3. Extension to Wiener's Attack. Recall that Wiener's attack on RSA applied when it was known that the private decryption exponent d was small, in the sense that

$$d \leq \frac{1}{3}N^{1/4}.$$

Boneh and Durfee, using an analogue of the bivariate case of Coppersmith's Theorem 17.3, extended Wiener's attack to the case where

$$d \leq N^{0.292};$$

using a heuristic algorithm. We do not go into the details but show how Boneh and Durfee proceed to a problem known as the small inverse problem.

Suppose we have an RSA modulus N, with encryption exponent e and decryption exponent d. By definition there is an integer k such that

$$ed + \frac{k\Phi}{2} = 1,$$

where $\Phi = \phi(N)$. Expanding the definition of Φ we find

$$ed + k\left(\frac{N+1}{2} - \frac{p+q}{2}\right) = 1.$$

We set

$$s = -\frac{p+q}{2},$$
$$A = \frac{N+1}{2}.$$

Then finding d, where d is small say $d < N^\delta$, is equivalent to finding the two small solutions k and s to the following congruence

$$f(k,s) = k(A+s) = 1 \pmod{e}.$$

To see that k and s are small relative to the modulus e for the above equation, notice that $e \approx N$ since d is small, and so

$$|s| < 2N^{0.5} \approx e^{0.5} \text{ and } |k| < \frac{2de}{\Phi} \leq \frac{3de}{N} \approx e^\delta.$$

We can interpret this problem as finding an integer which is close to A whose inverse is small modulo e. This is called the small inverse problem. Boneh and Durfee show that this problem has a solution when $\delta \leq 0.292$, hence extending the result of Wiener's attack. This is done by applying the multivariate analogue of Coppersmith's method to the polynomial $f(k,s)$.

5. Partial Key Exposure Attacks

Partial key exposure is related to the following question: Suppose in some cryptographic scheme the attacker recovers a certain set of bits of the private key, can the attacker use this to recover the whole private key? In other words, does partial exposure of the key result in a total break of the system? We shall present a number of RSA examples, however these are not the only ones. There are a number of results related to partial key exposure which relate to other schemes such as DSA or symmetric key based systems.

5.1. Partial Exposure of the MSBs of the RSA Decryption Exponent. Somewhat surprisingly for RSA, in the more common case of using a small public exponent e, one can trivially recover half of the bits of the private key d, namely the most significant, as follows. Recall there is a value of k such that $0 < k < e$ with

$$ed - k(N - (p+q) + 1) = 1.$$

Now suppose for each possible value of i, $0 < i \leq e$, the attacker computes

$$d_i = \lfloor (iN+1)/e \rfloor.$$

Then we have

$$|d_k - d| \leq k(p+q)/e \leq 3k\sqrt{N}/e < 3\sqrt{N}.$$

Hence, d_k is a good approximation for the actual value of d.

Now when $e = e$ it is clear that k, so we must have $k = 2$ and so d_2 reveals half of the most significant bits of d. Unluckily for the attack, and luckily for the user, there is no known way to recover the rest of d given only the most significant bits.

5.2. Partial Exposure of some bits of the RSA Prime Factors. Suppose our n-bit RSA modulus N is given by $p \cdot q$, with $p \approx q$ and that the attacker has found the $n/4$ least significant bits of p. Recall that p is only around $n/2$-bits long in any case, so this means the attacker is given the lower half of all the bits making up p. We write

$$p = x_0 2^{n/4} + p_0.$$

We then have, writing $q = y_0 2^{n/4} + q_0$,

$$N = p_0 q_0 \pmod{2^{n/4}}.$$

Hence, we can determine the value of q_0. We now write down the polynomial

$$\begin{aligned} p(x, y) &= (p_0 + 2^{n/4} x)(q_0 + 2^{n/4} y) \\ &= p_0 q_0 + 2^{n/4}(p_0 y + q_0 x) + 2^{n/2} xy. \end{aligned}$$

Now $p(x, y)$ is a bivariate polynomial of degree two which has known small solution modulo N, namely (x_0, y_0) where $0 < x_0, y_0 \leq 2^{n/4} \approx N^{1/4}$. Hence, using the heuristic bivariate extension of Coppersmith's Theorem 17.3, we can recover x_0 and y_0 in polynomial time and so factor the modulo N.

A similar attack applies when the attacker knows the $n/4$ most significant bits of p.

5.3. Partial Exposure of the LSBs of the RSA Decryption Exponent. We now suppose we are given, for small public exponent e, a quarter of the least significant bits of the private exponent d. That is we have d_0 where

$$d = d_0 + 2^{n/4} x_0$$

where $0 \leq x_0 \leq 2^{3n/4}$. Recall there is an even value of k with $0 < k < e$ such that

$$ed - k(N - (p + q) + 1) = 1.$$

We then have, since $N = pq$,

$$edp - kp(N - p + 1) + kN = p.$$

If we set $p_0 = p \pmod{2^{n/4}}$ then we have the equation

$$ed_0 p_0 - kp_0(N - p_0 + 1) + kN - p_0 = 0 \pmod{2^{n/4}}. \tag{15}$$

This gives us the following algorithm to recover the whole of d. For each value of k less than e we solve Equation (15) modulo $2^{n/4}$ for p_0. Each value of k will result in $O(\frac{n}{4})$ possible values for p_0. Using each of these values of p_0 in turn, we apply the previous technique for factoring N from Section 5.2. One such value of p_0 will be the correct value of $p \pmod{2^{n/4}}$ and so the above factorization algorithm will work and we can recover the value of d.

6. Fault Analysis

An interesting class of attacks results from trying to induce faults within a cryptographic system. We shall describe this area in relation to the RSA signature algorithm but similar attacks can be mounted on other cryptographic algorithms, both public and secret key.

Imagine we have a hardware implementation of RSA, in a smart card say. On input of some message m the chip will sign the message for us, using some internal RSA private key. The attacker wishes to determine the private key hidden within the smart card. To do this the attacker can try to make the card perform some of the calculation incorrectly, by either altering the card's environment by heating or cooling it or by damaging the circuitry of the card in some way.

An interesting case is when the card uses the Chinese Remainder Theorem to perform the signing operation, to increase efficiency, as explained in Chapter 15. The card first computes the hash of the message

$$h = H(m).$$

Then the card computes

$$s_p = h^{d_p} \pmod{p},$$
$$s_q = h^{d_q} \pmod{q},$$

where $d_p = d \pmod{p-1}$ and $d_q = d \pmod{q-1}$. The final signature is produced by the card from s_p and s_q via the Chinese Remainder Theorem using

- $u = (s_q - s_p)T \pmod{q}$,
- $s = s_p + up$,

where $T = p^{-1} \pmod{q}$.

Now suppose that the attacker can introduce a fault into the computation so that s_p is computed incorrectly. The attacker will then obtain a value of s such that

$$s^e \not\equiv h \pmod{p},$$
$$s^e = h \pmod{q}.$$

Hence, by computing

$$q = \gcd(s^e - h, N)$$

we can factor the modulus.

Chapter Summary

- Using a small private decryption exponent in RSA is not a good idea due to Wiener's attack.
- Lattices are discrete analogues of vector spaces, as such they have a shortest non-zero vector.
- Lattice basis reduction often allows us to find the shortest non-zero vector in a given lattice.
- Coppersmith's Theorem allows us to solve a modular polynomial equation when the solution is known to be small. The method works by building a lattice depending on the polynomial and then applying lattice basis reduction to obtain short vectors within this lattice.
- The Hastad and Franklin–Reiter attacks imply that one needs to be careful when designing padding schemes for RSA.
- Using Coppersmith's Theorem one can show that the revealing of a proportion of the bits of either p, q or d in RSA can lead to a complete break of the system.

Further Reading

The main paper on Coppersmith's lattice based attacks on RSA is by Coppersmith himself. Coppersmith's approach has itself been simplified somewhat in the paper of Howgrave-Graham. Our treatment of attacks on RSA in this chapter has closely followed the survey article by Boneh. A complete survey of lattice based methods in cryptography is given in the survey article by Nguyen and Stern.

D. Boneh. *Twenty years of attacks on the RSA cryptosystem.* Notices of the American Mathematical Society (AMS), **46**, 203–213, 1999.

D. Coppersmith. *Small solutions to polynomial equations, and low exponent RSA vulnerabilities.* J. Cryptology, **10**, 233–260, 1997.

N. Howgrave-Graham. *Finding small solutions of univariate modular equations revisited.* In Cryptography and Coding, Springer-Verlag LNCS 1355, 131–142, 1997.

P. Nguyen and J. Stern. *The two faces of lattices in cryptology.* In CALC '01, Springer-Verlag LNCS 2146, 146–180, 2001.

CHAPTER 18

Definitions of Security

Chapter Goals

- To explain the various notions of security of encryption schemes, especially semantic security and indistinguishability of encryptions.
- To explain the various attack notions, in particular the adaptive chosen ciphertext attack.
- To show how the concept of non-malleability and adaptive chosen ciphertext attacks are related.
- To examine the RSA and ElGamal encryption schemes in terms of these definitions of security and attacks.
- To explain notions related to the security of signature schemes.
- To show that naive (unhashed) RSA signatures are not secure due to the homomorphic property.

1. Security of Encryption

Up to now we have not looked much at how cryptographic primitives are turned into real protocols and systems. In designing cryptographic systems we need to understand exactly what security properties are guaranteed by the primitives and how the schemes can be made secure against all possible attacks.

Essentially, a cryptographic primitive is something like the raw RSA function or the modular exponentiation operation. Both are one-way functions, the RSA function being a trap-door one-way function and modular exponentiation being a one-way function. One needs to be careful how one uses these primitives in a real scheme. For example the raw RSA function is completely deterministic and so if used naively will always result in identical ciphertexts if you encrypt the same plaintext twice under the same key. This is a bad idea as we saw in Chapters 11 and 17. But there is a more fundamental reason to avoid deterministic public key encryption algorithms; to see why you need to consider what it means for an encryption scheme to be secure.

There are three things we need to define

- the goal of the adversary,
- the types of attack allowed,
- the computational model.

Our discussion will be in the context of public key encryption, but similar notions hold for symmetric ciphers, see Chapter 21.

1.1. Notions of Security. There are essentially three notions of security which we need to understand,

- perfect security,
- semantic security,
- polynomial security.

We shall discuss each of these notions in turn. These notions are far stronger than the simple notion of either recovering the private key or determining the plaintext which we have considered previously.

1.1.1. *Perfect Security:* Recall that a scheme is said to have perfect security, or to have information theoretic security, if an adversary with infinite computing power can learn nothing about the plaintext given the ciphertext. Shannon's Theorem, Theorem 5.4, essentially states that this means that the key is as long as the message, and the same key is never used twice.

The problem is that such systems cannot exist in the public key model, since the encryption key is assumed to be used for many messages and is usually assumed to be very short.

1.1.2. *Semantic Security:* Semantic security is like perfect security but we only allow an adversary with polynomially bounded computing power. Formally, for all probability distributions on the message space whatever an adversary can compute in polynomial time about the plaintext given the ciphertext, they should also be able to compute without the ciphertext. In other words, having the ciphertext does not help finding anything about the message.

The following is a simplified formal definition: Suppose that the information we wish to compute on the message space is a single bit, i.e. there is some function

$$g : M \longrightarrow \{0, 1\}.$$

We assume that over the whole message space we have

$$\Pr(g(m) = 1) = \Pr(g(m) = 0) = \frac{1}{2},$$

and that the plaintexts and ciphertexts are all of the same length (so for example the length of the ciphertext reveals nothing about the underlying plaintext).

The adversary we shall model as an algorithm S which on input of a public key y and a ciphertext c, encrypted under the public key y, will attempt to produce an evaluation of the function g on the plaintext for which c is the associated ciphertext. The output of S will therefore be a single bit corresponding to the value of g.

The adversary is deemed to be successful if the probability of it being correct is greater than one half. Clearly the adversary could always just guess the bit without seeing this ciphertext, hence we are saying a successful adversary is one which can do better after seeing the ciphertext. We therefore define the advantage of the adversary S as

$$\mathrm{Adv}_S = \left| \Pr(S(c, y) = g(d_x(c))) - \frac{1}{2} \right|,$$

where d_x denotes decryption under the private key x associated to the public key y. A scheme is then said to be semantically secure if we have

$$\mathrm{Adv}_S \leq \frac{1}{p(k)},$$

for all adversaries S, all functions g and all polynomial functions $p(k)$ when the security parameter k is sufficiently large.

1.1.3. *Polynomial Security:* The trouble with the definition of semantic security is that it is hard to show that a given encryption scheme has this property. Polynomial security, sometimes called indistinguishability of encryptions, is a much easier property to show that a given system has. Luckily, we will show that if a system has polynomial security then it has semantic security. Hence, to show a system is semantically secure all we need do is show that it is polynomially secure.

A system is said to have indistinguishable encryptions, or to have polynomial security, if no adversary can win the following game with probability greater than one half. The adversary A is given a public key encryption function f_y corresponding to some public key y. The adversary now runs in two stages:

- **Find**: In the find stage the adversary produces two plaintext messages m_0 and m_1, of equal length.
- **Guess**: The adversary is now given the encryption c_b of one of the plaintexts m_b for some secret hidden bit b. The goal of the adversary is to now guess the value of b with probability greater than one half.

Note, this means that an encryption function which is polynomially secure must be probabilistic in nature, otherwise in the guess stage the adversary can compute

$$c_1 = f_y(m_1)$$

and test whether this is equal to c_b or not. The advantage of an adversary A is therefore defined to be

$$\mathrm{Adv}_A = \left| \Pr\left(A(\mathbf{guess}, c_b, y, m_0, m_1) = b\right) - \frac{1}{2} \right|,$$

since the adversary A could always simply guess the hidden bit b. A scheme is then said to be polynomially secure if we have

$$\mathrm{Adv}_A \leq \frac{1}{p(k)},$$

for all adversaries A and all polynomials p and sufficiently large k.

1.2. Notions of Attacks. At this stage we need to introduce various attack models. There are three basic attack models:

- passive attack, sometimes called a chosen plaintext attack, often denoted CPA,
- chosen ciphertext attack, often denoted CCA1,
- adaptive chosen ciphertext attack, often denoted CCA2.

1.2.1. *Passive Attack:* A passive attack is a very weak form of attack. The adversary Eve is allowed to look at various encrypted messages. She also has access to a (black) box which performs encryption but not decryption. Hence this models a simple attack on a public key system, since in a public key system everyone, including the attacker, has access to the encryption function.

1.2.2. *Chosen Ciphertext Attack:* A chosen ciphertext attack (CCA1) is often called a lunch-time attack and represents a slightly stronger form of attack. Eve is now given access to a black box which performs decryptions. Eve can ask the box to decrypt a polynomial number of ciphertexts of her choosing, at lunch-time whilst the real user of the box is away at lunch. After this, at some point in the afternoon, she is given a target ciphertext and asked to decrypt this, or find information about the underlying plaintext, on her own, without using the box.

In the context of our game of polynomial security given above, this means that the adversary can query the decryption box during the **find** stage of the algorithm but not the **guess** stage.

1.2.3. *Adaptive Chosen Ciphertext Attack:* An adaptive chosen ciphertext attack (CCA2) is a very strong form of attack. Eve is now allowed to ask the decryption box to decrypt any ciphertext of her choosing, bar the target ciphertext. It is widely regarded that any new proposed public key encryption algorithm should meet the requirements of polynomial security under an adaptive chosen ciphertext attack.

Hence, in our game of polynomial security, the adversary can in both the **find** and **guess** stages ask the decryption box to decrypt any ciphertext, except they are not allowed to ask the box to decrypt the challenge c_b, since otherwise the game would be far too easy.

The following definition is the accepted definition of what it means for a public encryption scheme to be secure, a similar notion applies to symmetric key systems.

DEFINITION 18.1. *A public key encryption algorithm is said to be secure if it is* semantically secure *against an adaptive chosen ciphertext attack.*

However, usually it is easier to show security under the following definition.

DEFINITION 18.2. *A public key encryption algorithm is said to be secure if it is* polynomially secure *against an adaptive chosen ciphertext attack.*

These two notions are however related. For example, we shall now show the following important result:

THEOREM 18.3. *For a passive adversary, a system which is polynomially secure must necessarily be semantically secure.*

PROOF. We proceed by contradiction. Assume that there is an encryption algorithm which is not semantically secure, i.e. there is an algorithm S with

$$\text{Adv}_S > \frac{1}{p(k)}$$

for a polynomial $p(k)$ and all sufficiently large values of k. We shall show that it is then not polynomial secure either. To do this we construct an adversary A against the polynomial security of the scheme, which uses the adversary S against the semantic security as an oracle.

The **find** stage of adversary A outputs two messages m_0 and m_1 such that

$$g(m_0) \neq g(m_1).$$

Such messages will be easy to find given our earlier simplified formal definition of semantic security, since the output of g is equiprobable over the whole message space.

The adversary A is then given an encryption c_b of one of these and is asked to determine b. In the **guess** stage the adversary passes the ciphertext c_b to the oracle S. The oracle S returns with its best guess as to the value of

$$g(m_b).$$

The adversary A can now compare this value with that of $g(m_0)$ and $g(m_1)$ and hence output a guess as to what the value of b is.

Clearly if S is successful in breaking the semantic security of the scheme, then A will be successful in breaking the polynomial security. So

$$\Pr\left(A(\textbf{guess}, c_b, y, m_0, m_1) = b\right) = \Pr\left(S(c, y) = g(d_x(c))\right).$$

So

$$\text{Adv}_A = \text{Adv}_S > \frac{1}{p(k)}.$$

Hence, polynomial security must imply semantic security. □

Note, with a more complicated definition of semantic security it can be shown that for passive adversaries the notions of semantic and polynomial security are equivalent.

1.3. Other Security Concepts.

1.3.1. *Non-Malleability:* An encryption scheme is said to be non-malleable if given a ciphertext C, corresponding to an unknown plaintext M, it is impossible to determine a valid ciphertext C' on a 'related' message M'. Note, that 'related' is defined vaguely here on purpose. Non-malleability is important due to the following result, for which we only give an informal proof based on our vague definition of non-malleability. A formal proof can however be given, with an associated formal definition of non-malleability.

LEMMA 18.4. *A malleable encryption scheme is not secure against an adaptive chosen ciphertext attack.*

PROOF. Suppose the scheme is malleable, then given the target ciphertext c_b we alter it to a related one, namely ${c_b}'$. The relation should be such that there is also a known relationship between ${m_b}'$ and m_b. Then the adversary can ask the decryption oracle to decrypt ${c_b}'$ to reveal ${m_b}'$, an oracle query which is allowed in our prior game. Then from ${m_b}'$ the adversary can recover m_b. □

Later we shall show that almost all the public key schemes we have seen so far are malleable. However, it is known that a scheme which is non-malleable against a CCA2 attack is also polynomially secure against a CCA2 attack and viceversa. Hence, a non-malleable encryption scheme will meet our previous definition of security of a public key encryption scheme.

1.3.2. *Plaintext Aware:* If a scheme is plaintext aware then we have a very strong notion of security. A scheme is called plaintext aware if it is computationally difficult to construct a valid ciphertext without being given the corresponding plaintext to start with. Hence, plaintext awareness implies one cannot mount a CCA attack. Since to write down a ciphertext requires you to know the plaintext, so why would you ask the decryption oracle to decrypt the ciphertext?

Plaintext awareness has only been defined within the context of the *random oracle model* of computation. In this model one assumes that idealized hash functions exist; these are one-way functions which:

- take polynomial time to evaluate,
- to an observer cannot be distinguished from random functions.

The random oracle model occurs in a number of proofs of security. These proofs tell us nothing about the real-world security of the scheme under consideration. But they do show that any real-world attack must make use of the actual definition of the hash function deployed. The usual state of affairs is that we prove a protocol secure in the random oracle model and then make a real-life protocol by replacing the random oracle with a hash function like SHA-1 or MD5.

2. Security of Actual Encryption Algorithms

In this section we show that neither the RSA or ElGamal encryption algorithms meet our stringent goals of semantic security against an adaptive chosen ciphertext attack. This should surprise you since we have claimed that RSA is one of the most used and important algorithms around. But we have not shown you how RSA is actually used in practice. So far we have only given the simple mathematical description of the algorithms. As we progress onwards the mathematics of public key encryption gets left behind and becomes replaced by cryptographic engineering.

To make the discussion easier we will use the fact that semantic security and indistinguishability of encryptions (polynomial security) are equivalent.

2.1. RSA.

LEMMA 18.5. *RSA is not polynomially secure.*

PROOF. Suppose the attacker knows that the user only encrypts one of two messages

$$m_1 \text{ or } m_2.$$

These could be buy or sell, yes or no etc. The attacker is assumed to know the user's public key, namely N and e. On receiving the ciphertext c the attacker wants to determine whether the corresponding plaintext m was equal to m_1 or m_2. All the adversary need do is compute

$$c' = m_1^e \pmod{N}.$$

Then

- if $c' = c$ then the attacker knows that $m = m_1$,
- if $c' \neq c$ then the attacker knows that $m = m_2$.

□

The problem is that the attacker has access to the encryption function, it is a public key scheme after all.

Now suppose that decryption was not injective, that each plaintext could correspond to a large number of ciphertexts. The exact ciphertext produced being determined by the encryption function at runtime. In other words the encryption algorithm should be probabilistic in nature and not just deterministic. Later we shall see a variant of RSA which has this property and so the attack described in the above proof does not apply. But using a deterministic encryption function is not the only problem with RSA.

Essentially RSA is malleable due to the homomorphic property.

DEFINITION 18.6 (Homomorphic Property). *Given the encryption of m_1 and m_2 we can determine the encryption of $m_1 \cdot m_2$, without knowing m_1 or m_2.*

That RSA has the homomorphic property follows from the equation

$$(m_1 \cdot m_2)^e \pmod N = ((m_1{}^e \pmod N) \cdot (m_2{}^e \pmod N)) \pmod N.$$

One can use the homomorphic property to show that RSA is insecure under an adaptively chosen ciphertext attack.

LEMMA 18.7. *RSA is not CCA2 secure.*

PROOF. Suppose the message Eve wants to break is

$$c = m^e \pmod N.$$

Eve creates the 'related' ciphertext $c' = 2^e c$ and asks her oracle to decrypt c' to give m'. Eve can then compute

$$\begin{aligned} \frac{m'}{2} &= \frac{c'^d}{2} = \frac{(2^e c)^d}{2} \\ &= \frac{2^{ed} c^d}{2} = \frac{2m}{2} = m. \end{aligned}$$

□

2.2. ElGamal. Recall the decision Diffie–Hellman (known as DDH) assumption for a group $G = \langle g \rangle$ is, given g^x, g^y and g^z, determine whether

$$x \cdot y = z \pmod{\#G}.$$

LEMMA 18.8. *If DDH is hard in the group G then ElGamal encryption is polynomially secure against a passive adversary.*

PROOF. To show that ElGamal encryption is polynomially secure assuming that DDH holds we first assume that we have a polynomial-time algorithm A which breaks the polynomial security of ElGamal encryption. Then using this algorithm A as a subroutine we give an algorithm to solve the DDH problem. This is similar to the technique used in an earlier chapter where we showed that an algorithm which could break ElGamal encryption, in the sense of decrypting a ciphertext to reveal a plaintext, could be used to solve the computational Diffie–Hellman problem. Now with our stronger definition of security, namely polynomial security, we can only relate breaking the system to the solution of a (supposedly) easier problem.

Recall A should run in two stages:

- A **find** stage which outputs two messages and some state information and which takes a public key as input.
- A **guess** stage which takes as input a ciphertext, a public key, two messages and some state information and guesses which plaintext the ciphertext corresponds to.

In addition, recall the ElGamal ciphertext is of the form

$$(g^k, m \cdot h^k)$$

where

- k is an ephemeral per message secret,
- h is the public key.

Our algorithm for solving DDH now proceeds as in Algorithm 18.1

To see why this algorithm solves the DDH problem consider the following argument.

- In the case when $z = x \cdot y$ then the encryption input into the guess stage of algorithm A will be a valid encryption of m_b. Hence, if algorithm A can really break the semantic security of ElGamal encryption then the output b' will be correct and the above algorithm will return **true**.
- Now suppose that $z \neq x \cdot y$ then the encryption input into the guess stage is almost definitely invalid, i.e. not an encryption of m_1 or m_2. Hence, the output b' of the guess stage will be independent of the value of b. Therefore we expect the above algorithm to return **true** or **false** with equal probability.

Algorithm 18.1: Algorithm to solve DDH given an algorithm to break the semantic security of ElGamal

As input we have g^x, g^y and g^z
$h = g^x$
$(m_0, m_1, s) = A(\textbf{find}, h)$
$c_1 = g^y$
Choose b randomly from $\{0, 1\}$
$c_2 = m_b \cdot g^z$
$b' = A(\textbf{guess}, (c_1, c_2), h, m_0, m_1, s)$
if $b = b'$ **then return true**
else return false

If we repeat the Algorithm 18.1 a few times, then we obtain a probabilistic polynomial-time algorithm to solve the DDH problem. But we have assumed that no such algorithm exists, so this implies that our algorithm A cannot exist either. So the DDH assumption implies that no adversary against the polynomial security of ElGamal encryption under chosen plaintext attacks can exist. □

However although it is semantically secure against chosen plaintext attacks, ElGamal encryption is trivially malleable.

LEMMA 18.9. *ElGamal encryption is malleable.*

PROOF. Suppose Eve sees the ciphertext

$$(c_1, c_2) = (g^k, m \cdot h^k).$$

She can then create a valid ciphertext of the message $2 \cdot m$ without ever knowing m, nor the ephemeral key k nor the private key x. The ciphertext she can produce is given by

$$(c_1, 2 \cdot c_2) = (g^k, 2 \cdot m \cdot h^k).$$

□

One can use this malleability property, just as we did with RSA, to show that ElGamal encryption is insecure under an adaptively chosen ciphertext attack.

LEMMA 18.10. *ElGamal is not CCA2 secure.*

PROOF. Suppose the message Eve wants to break is

$$c = (c_1, c_2) = (g^k, m \cdot h^k).$$

Eve creates the related message

$$c' = (c_1, 2c_2)$$

and asks her decryption oracle to decrypt c' to give m'. Then Eve computes

$$\frac{m'}{2} = \frac{2c_2 c_1^{-x}}{2} = \frac{2mh^k g^{-xk}}{2} = \frac{2mg^{xk}g^{-xk}}{2} = \frac{2m}{2} = m.$$

□

3. A Semantically Secure System

We have seen that RSA is not semantically secure even against a passive attack; it would be nice to give a system which is semantically secure and is based on some factoring-like assumption. Historically the first system to meet these goals was one by Goldwasser and Micali, although this is not used in real-life applications. This scheme's security is based on the hardness of the QUADRES problem, namely given a composite integer N it is hard to test whether a is a quadratic residue or not without knowledge of the factors of N.

Let us recap that the set of squares in $(\mathbb{Z}/N\mathbb{Z})^*$ is denoted by

$$Q_N = \{x^2 \pmod N : x \in (\mathbb{Z}/N\mathbb{Z})^*\},$$

and J_N denotes the set of elements with Jacobi symbol equal to plus one, i.e.

$$J_N = \left\{a \in (\mathbb{Z}/N\mathbb{Z})^* : \left(\frac{a}{N}\right) = 1\right\}.$$

The set of pseudo-squares is the difference $J_N \setminus Q_N$. For an RSA-like modulus $N = p \cdot q$ the number of elements in J_N is equal to $(p-1)(q-1)/2$, whilst the number of elements in Q_N is $(p-1)(q-1)/4$. The QUADRES problem is that given an element x of J_N, it is hard to tell whether $x \in Q_N$, whilst it is easy to tell if $x \in J_N$ or not.

We can now explain the Goldwasser–Micali encryption system.

3.0.1. *Key Generation:* As a private key we take two large prime number p and q and then compute the public modulus

$$N = p \cdot q.$$

The public key also contains an integer

$$y \in J_N \setminus Q_N.$$

The value of y is computed by the public key owner by first computing elements $y_p \in \mathbb{F}_p^*$ and $y_q \in \mathbb{F}_q^*$ such that

$$\left(\frac{y_p}{p}\right) = \left(\frac{y_q}{q}\right) = -1.$$

Then the value of y is computed from y_p and y_q via the Chinese Remainder Theorem. A value of y computed in this way clearly does not lie in Q_N, but it does lie in J_N since

$$\left(\frac{y}{N}\right) = \left(\frac{y}{p}\right) \cdot \left(\frac{y}{q}\right) = \left(\frac{y_p}{p}\right) \cdot \left(\frac{y_q}{q}\right) = (-1) \cdot (-1) = 1.$$

3.0.2. *Encryption:* The Goldwasser–Micali encryption system encrypts one bit of information at a time. To encrypt the bit b,

- pick $x \in (\mathbb{Z}/N\mathbb{Z})^*$ at random,
- compute $c = y^b x^2 \pmod N$.

The ciphertext is then the value of c. Notice that this is very inefficient since a single bit of plaintext requires $\log_2 N$ bits of ciphertext to transmit it.

3.0.3. *Decryption:* Notice that the ciphertext c will always be an element of J_N. However, if the message bit b is zero then the value of c will be a quadratic residue, otherwise it will be a quadratic non-residue. So all the decryptor has to do to recover the message is determine whether c is a quadratic residue or not modulo N. But the decryptor is assumed to know the factors of N and so can compute the Legendre symbol

$$\left(\frac{c}{p}\right).$$

If this Legendre symbol is equal to plus one then c is a quadratic residue and so the message bit is zero. If however the Legendre symbol is equal to minus one then c is not a quadratic residue and so the message bit is one.

3.0.4. *Proof of Security:* We wish to show that the Goldwasser–Micali encryption system is secure against passive adversaries assuming that the QUADRES problem is hard for the modulus N.

LEMMA 18.11. *If the QUADRES problem is hard for the modulus N then the above encryption system is polynomially secure against passive adversaries.*

PROOF. To do this we assume we have such an adversary A against the above encryption scheme of Goldwasser and Micali. We will now show how to use this adversary to solve a QUADRES problem.

Suppose we are given $j \in J_N$ and we are asked to determine whether $j \in Q_N$. Since our system only encrypts bits the **find** stage of the adversary A, on input of the public key (y, N), will simply output the two messages

$$m_0 = 0 \text{ and } m_1 = 1.$$

We now form the ciphertext

$$c = j.$$

Note that if j is a quadratic residue then this value of c will be a valid encryption of the message m_0, however if j is not a quadratic residue then this value of c will be a valid encryption of the message m_1. We therefore ask our adversary A to **guess** which message is c a valid encryption of. Hence, from the output of A we can decide whether the value of j is an element of Q_N or not. □

3.0.5. *Adaptive Adversaries:* Note the above argument says nothing about whether the Goldwasser–Micali encryption scheme is secure against adaptive adversaries. In fact one can show it is not secure against such adversaries.

LEMMA 18.12. *The Goldwasser–Micali encryption scheme is insecure against an adaptive chosen ciphertext attack.*

PROOF. Suppose c is the target ciphertext and we want to determine what bit b is, that c is a valid encryption of. Recall

$$c = y^b x^2 \pmod N.$$

Now the rules of the game do not allow us to ask our decryption oracle to decrypt c, but we can ask our oracle to decrypt any other ciphertext.

We therefore produce the ciphertext

$$c' = c \cdot z^2 \pmod N,$$

for some random value of $z \neq 0$. It is easy to see that c' is an encryption of the same bit b. Hence, by asking our oracle to decrypt c' we will obtain the decryption of c. □

4. Security of Signatures

There are a number of possible security notions for signature schemes. Just as with standard handwritten signatures we are interested in trying to stop the adversary forging a signature on a given message. The three main types of forgery are:

Total Break: The forger can produce signatures just as if he were the valid key holder. This is akin to recovering the private key and corresponds to the similar type of break of an encryption algorithm.

Selective Forgery: In this case the adversary is able to forge a signature on a single message of her choosing. This is similar to the ability of an adversary of an encryption algorithm being able to decrypt a message but not recover the private key.

Existential Forgery: In this case the adversary is able to forge a signature on a single message, which could just be a random bit string. You should think of this as being analogous to semantic security of encryption schemes.

In practice we usually want our schemes to be secure against an attempt to produce a selective forgery. But we do not know how the signature scheme is to be used in real life, for example it may be used in a challenge/response protocol where random bit strings are signed by various parties. Hence, it is prudent to insist that any signature scheme should be secure against an existential forgery.

Along with types of forgery we also have types of attack. The weakest attack is that of a passive attacker, who is simply given the public key and is then asked to produce a forgery, be it selective or existential. The strongest form of attacker is an adaptive active attacker, such an attacker is given access to a signing oracle which will produce valid signatures for the public key. The goal of an active attacker is to produce a signature on a message which they have not yet asked their signing oracle. This leads to the following definition:

DEFINITION 18.13. *A signature scheme is deemed to be secure if it is infeasible for an adaptive adversary to produce an existential forgery.*

We have already remarked that the use of hash functions is crucial in a signature algorithm. One can see this again by considering a raw RSA signature algorithm defined by

$$s = m^d \pmod{N}.$$

It is trivial to produce an existential forgery with such a scheme via a passive attack. The attacker picks s at random and then computes

$$m = s^e \pmod{N}.$$

The attacker then has the signature s on the message m.

With such a scheme it is also very easy for an active attacker to produce a selective forgery: Suppose the attacker wishes to produce a signature s on the message m. They first generate a random $m_1 \in (\mathbb{Z}/N\mathbb{Z})^*$ and compute

$$m_2 = \frac{m}{m_1}.$$

Then the attacker asks her oracle to sign the messages m_1 and m_2. This results in two signatures s_1 and s_2 such that

$$s_i = {m_i}^d \pmod{N}.$$

The attacker can then compute the signature on the message m by computing

$$s = s_1 \cdot s_2 \pmod{N}$$

since

$$\begin{aligned} s &= s_1 \cdot s_2 \pmod{N} \\ &= {m_1}^d \cdot {m_2}^d \pmod{N} \\ &= (m_1 \cdot m_2)^d \pmod{N} \\ &= m^d \pmod{N}. \end{aligned}$$

Chapter Summary

- The definition of what it means for a scheme to be secure can be different from one's initial naive view.
- Today the notion of semantic security is the de facto standard definition for encryption schemes.
- Semantic security is hard to prove but it is closely related to the simpler notion of polynomial security, often called indistinguishability of encryptions.
- We also need to worry about the capabilities of the adversary. For encryption this is divided into three categories: chosen plaintext attacks, chosen ciphertext attacks and adaptive chosen ciphertext attacks.
- Some schemes, such as ElGamal encryption and the Goldwasser–Micali system, are polynomially secure against passive adversaries, but not against active adaptive adversaries. Others, such as RSA, are not even polynomially secure against passive adversaries.
- Security against adaptive adversaries and the notion of non-malleability are closely related.
- Similar considerations apply to the security of signature schemes, where we are now interested in the notion of existential forgery under an active attack.

Further Reading

A good introduction to the definitional work in cryptography based on provable security and its extensions and foundations in the idea of zero-knowledge proofs can be found in the book by Goldreich. A survey of the initial work in this field, up to around 1990, can be found in the article by Goldwasser.

O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudo-randomness.* Springer-Verlag, 1999.

S. Goldwasser. *The Search for Provable Secure Cryptosystems.* In Cryptology and Computational Number Theory, Proc. Symposia in Applied Maths, Volume 42, 1990.

CHAPTER 19

Complexity Theoretic Approaches

Chapter Goals

- To introduce the concepts of complexity theory needed to study cryptography.
- To understand why complexity theory on its own cannot lead to secure cryptographic systems.
- To explain the Merkle–Hellman system and why it is weak.
- To introduce the concept of bit security.
- To introduce the idea of random self-reductions.

1. Polynomial Complexity Classes

A common mistake in building new cryptographic schemes is to pick hard problems which are only hard on certain problem instances and are not hard for an average instance. To elaborate on this in more detail we need to recap some basic ideas from complexity theory.

Recall a decision problem $\mathcal{D}P$ is a problem with a yes/no answer, which has inputs (called instances) I coded in some way (for example as a binary string of some given size n). Often one has a certain subset S of instances in mind and one is asking 'Is $I \in S$?'. For example one could have

- I is the set of all integers, S is the subset of primes. Hence the decision problem is: Given an integer say whether it is prime or not.
- I is the set of all graphs, S is the subset of all graphs which are colourable using k colours only. Hence the decision problem is: Given a graph tell me whether it is colourable using only k colours. Recall a graph consisting of vertices V and edges E is colourable by k colours if one can assign k colours (or labels) to each vertex so that no two vertices connected by an edge share the same colour.

Whilst we have restricted ourselves to decision problems, one can often turn a standard computational problem into a decision problem. As an example of this consider the cryptographically important knapsack problem.

DEFINITION 19.1 (Decision Knapsack Problem). *Given a pile of n items, each with different weights w_i, is it possible to put items into a knapsack to make a specific weight S? In other words, do there exist $b_i \in \{0, 1\}$ such that*

$$S = b_1 w_1 + b_2 w_2 + \cdots + b_n w_n?$$

Note, the time taken to solve this seems to grow in the worst case as an exponential function in terms of the number of weights.

As stated above the knapsack problem is a decision problem but we could ask for an algorithm to actually find the values b_i.

DEFINITION 19.2 (Knapsack Problem). *Given a pile of n items, each with different weights w_i, is it possible to put items into a knapsack to make a specific weight S? If so can one find the $b_i \in \{0,1\}$ such that*

$$S = b_1 w_1 + b_2 w_2 + \cdots + b_n w_n?$$

We assume only one such assignment of weights is possible.

We can turn an oracle for the decision knapsack problem into one for the knapsack problem proper. To see this consider the Algorithm 19.1 which assumes an oracle $O(w_1, ..., w_n, S)$ for the decision knapsack problem.

Algorithm 19.1: Knapsack algorithm, assuming a decision knapsack oracle

```
if O(w_1, ..., w_n, S) =false then return (false)
T = S
b_1 = b_2 = ... = b_n = 0
for i = 1 to n do
    if T = 0 then return ((b_1, ..., b_n))
    if O(w_{i+1}, ..., w_n, T − w_i) =true then
        T = T − w_i
        b_i = 1
    end
end
```

A decision problem $\mathcal{D}P$ is said to lie in complexity class $\mathcal{P}$ if there is an algorithm which takes any instance I, for which the answer is *yes*, and delivers the answer, namely *yes*, in polynomial time. We measure time in terms of bit operations and polynomial time means the number of bit operations is bounded by some polynomial function of the input size of the instance I. If the answer to the instance is *no* then the algorithm is not even required to answer in polynomial time, but if it does answer then it should return the correct answer.

By replacing *yes* by *no* in the above paragraph we obtain the class $\text{co} - \mathcal{P}$. This is the set of problems for which there exists an algorithm which on input of an instance I, for which the answer is *no*, will output *no* in polynomial time.

LEMMA 19.3.

$$\mathcal{P} = \text{co} - \mathcal{P}.$$

PROOF. Suppose A is the algorithm which for a problem instance I outputs *yes* if the answer is *yes* in time n^c for some constant c.

We turn A into a polynomial-time algorithm to answer *no* if the answer is *no* in time $n^c + 1$. We simply run A; if it takes more than n^c steps then we terminate A and output *no*. □

The problems which lie in complexity class $\mathcal{P}$ are those for which we have an efficient solution algorithm. In other words these are the things which are easy to compute. For example

- Given integers x, y and z do we have $z = x \cdot y$, i.e. is multiplication easy?
- Given a ciphertext c, a key k and a plaintext m, is c the encryption of m under your favourite encryption algorithm?

Of course in the last example I have assumed your favourite encryption algorithm has an encryption/decryption algorithm which runs in polynomial time. If your favourite encryption algorithm is not of this form, then one must really ask how have you read so far in this book?

A decision problem lies in complexity class $\mathcal{NP}$, called non-deterministic polynomial time, if for every instance for which the answer is *yes*, there exists a witness for this which can be checked

in polynomial time. If the answer is *no* we again do not assume the algorithm terminates, but if it does it must do so by answering *no*. One should think of the witness as a proof that the instance I lies in the subset S.

Examples include

- The problem 'Is N composite?' lies in $\mathcal{NP}$ as one can give a non-trivial prime factor as a witness. This witness can be checked in polynomial time since division can be performed in polynomial time.
- The problem 'Is G k-colourable?' lies in $\mathcal{NP}$ since as a witness one can give the colouring.
- The problem 'Does this knapsack problem have a solution?' lies in $\mathcal{NP}$ since as a witness we can give the values b_i.

Note, in none of these examples have we assumed the witness itself can be computed in polynomial time, only that the witness can be checked in polynomial time. Note that we trivially have

$$\mathcal{P} \subset \mathcal{NP}.$$

The main open problem in theoretical computer science is the question does $P = \mathcal{NP}$? Most people believe in the conjecture

CONJECTURE 19.4.

$$P \neq \mathcal{NP}.$$

The set $\text{co} - \mathcal{NP}$ is defined in the same way that the class $\text{co} - \mathcal{P}$ was derived from $\mathcal{P}$. The class $\text{co} - \mathcal{NP}$ is the set of problems for which a witness exists for every instance with a *no* response which can be checked in polynomial time. Unlike the case of $\text{co} - \mathcal{P}$ and $\mathcal{P}$ we have

$$\text{If } \mathcal{P} \neq \mathcal{NP} \text{ then } \mathcal{NP} \neq \text{co} - \mathcal{NP}.$$

Hence, one should assume that $\mathcal{NP} \neq \text{co} - \mathcal{NP}$.

One can consider trying to see how small a witness for being in class $\mathcal{NP}$ can be found. For example consider the problem `COMPOSITES`. Namely given $N \in \mathbb{Z}$ determine whether N is composite. As we remarked earlier this clearly lies in class $\mathcal{NP}$. But a number N can be proved composite in the following ways:

- Giving a factor. In this case the size of the witness is
$$O(\log n).$$
- Giving a Miller–Rabin witness a. Now, assuming the Generalized Riemann Hypothesis (GRH) the size of the witness is
$$O(\log \log n)$$
since we have
$$a \leq O((\log n)^2).$$

A decision problem $\mathcal{D}P$ is said to be $\mathcal{NP}$-complete if every other problem in class $\mathcal{NP}$ can be reduced to this problem in polynomial time. In other words we have

$$\mathcal{D}P \in \mathcal{P} \text{ implies } \mathcal{P} = \mathcal{NP}.$$

In some sense the $\mathcal{NP}$-complete problems are the hardest problems for which it is feasible to ask for a solution. There are a huge number of $\mathcal{NP}$-complete problems of which the two which will interest us are

- the 3-colouring problem,
- the knapsack problem.

We know factoring (or COMPOSITES) lies in $\mathcal{NP}$ but it is unknown whether it is $\mathcal{NP}$-complete. In fact it is a widely held view that all the hard problems on which cryptography is based, e.g. factoring, discrete logarithms etc., are not related to an $\mathcal{NP}$-complete problem even though they lie in class $\mathcal{NP}$.

From this we can conclude that factoring is not a very difficult problem at all, not at least compared with the knapsack problem or the 3-colouring problem. So why do we not use $\mathcal{NP}$-complete problems on which to base our cryptographic schemes? These are after all a well-studied set of problems for which we do not expect there ever to be an efficient solution.

However, this approach has had a bad track record, as we shall show later when we consider the knapsack based system of Merkle and Hellman. For now we simply mention that complexity theory, and so the theory of $\mathcal{NP}$-completeness, is about worst case complexity. But for cryptography we want a problem which, for suitably chosen parameters, is hard on average. It turns out that the knapsack problems that have in the past been proposed for use in cryptography are always 'average' and efficient algorithms can always be found to solve them.

We end this section by illustrating this difference between hard and average problems using the k-colouring problem, when $k = 3$. Although determining whether a graph is 3-colourable is in general (in the worst case) $\mathcal{NP}$-complete, it is very easy on average. This is because the average graph, no matter how large it is, will not be 3-colourable. In fact, for almost all input graphs the following algorithm will terminate saying that the graph is not 3-colourable in a constant number of iterations.

- Take a graph G and order the vertices in any order $v_1, \ldots, v_t$.
- Call the colours $\{1, 2, 3\}$.
- Now traverse the graph in the order of the vertices just decided.
- On visiting a new vertex use the smallest possible colour (i.e. one from the set $\{1, 2, 3\}$ which does not appear as the colour of an adjacent vertex).
- If you get stuck traverse back up the graph to the most recently coloured vertex and use the next colour available, then continue again.
- If at any point you run out of colours for the first vertex then terminate and say the graph is *not* 3-colourable.
- If you are able to colour the last vertex then terminate and output that the graph *is* 3-colourable.

The interesting thing about the above algorithm is that it can be shown that for a *random* graph of t vertices the average number of vertices travelled in the algorithm is less than 197 regardless of the number of vertices t in the graph.

2. Knapsack-Based Cryptosystems

One of the earliest public key cryptosystems was based on the knapsack, or subset sum, problem which was believed to be very hard in general to solve. In fact it is $\mathcal{NP}$-complete, however it turns out that this knapsack-based scheme, and almost all others, can be shown to be insecure.

The idea is to create two problem instances. A public one which is hard, which is believed to be a general knapsack problem, and a private problem which is easy. In addition there should be some private trapdoor information which transforms the hard problem into the easy one.

This is rather like the use of the RSA assumption. It is hard to extract eth roots modulo a composite number, but easy to extract eth roots modulo a prime number. Knowing the trapdoor information, namely the factorization of the RSA modulus, allows us to transform the hard problem into the easy problem. However, the crucial difference is that whilst factoring (for suitably chosen moduli) is hard on average, it is difficult to produce knapsack problems which are hard on average.

This is even though the general knapsack problem is considered harder than the general factorization problem.

Whilst the general knapsack problem is hard there is a particularly easy set of problems based on super-increasing knapsacks. A super-increasing knapsack problem is one where the weights are such that each one is greater than the sum of the preceding ones, i.e.

$$w_j > \sum_{i=1}^{j-1} w_i.$$

As an example one could take the set

$$\{2, 3, 6, 13, 27, 52\}$$

or one could take

$$\{1, 2, 4, 8, 16, 32, 64, \ldots\}.$$

Given a super-increasing knapsack problem, namely an ordered set of such super-increasing weights $\{w_1, \ldots, w_n\}$ and a target weight S, determining which weights to put in the sack is a linear operation, as can be seen from Algorithm 19.2.

Algorithm 19.2: Solving a super-increasing knapsack problem

```
for i = n downto 1 do
    if S >= w_i then
        b_i = 1 ;
        S = S − w_i ;
    end
    else  b_i = 0 ;
end
if S = 0 then  return (b_1, b_2, ..., b_n) ;
else return ("No Solution") ;
```

The Merkle–Hellman cryptosystem takes as a private key a super-increasing knapsack problem and from this creates (using a private transform) a so-called 'hard knapsack' problem. This hard problem is then the public key.

This transform is achieved by choosing two private integers N and M, such that

$$\gcd(N, M) = 1$$

and multiplying all values of the super-increasing sequence by $N \pmod{M}$. For example if we take as the private key

- the super-increasing knapsack $\{2, 3, 6, 13, 27, 52\}$,
- $N = 31$ and $M = 105$.

Then the associated public key is given by the 'hard' knapsack

$$\{62, 93, 81, 88, 102, 37\}.$$

We then publish the hard knapsack problem as our public key, with the idea that only someone who knows N and M can transform back to the easy super-increasing knapsack.

For Bob to encrypt a message to us, he first breaks the plaintext into blocks the size of the weight set. The ciphertext is then the sum of the weights where a bit is set. So for example if the message is given by

$$\text{Message} = 011000\ 110101\ 101110$$

Bob obtains, since our public knapsack is $\{62, 93, 81, 88, 102, 37\}$, that the ciphertext is

$$174, 280, 333,$$

since

- 011000 corresponds to $93 + 81 = 174$,
- 110101 corresponds to $62 + 93 + 88 + 37 = 280$,
- 101110 corresponds to $62 + 81 + 88 + 102 = 333$.

The legitimate recipient knows the private key N, M and $\{2, 3, 6, 13, 27, 52\}$. Hence, by multiplying each ciphertext block by $N^{-1} \pmod M$ the hard knapsack is transformed into the easy knapsack problem.

In our case $N^{-1} = 61 \pmod M$, and so the decryptor performs the operations

- $174 \cdot 61 = 9 = 3 + 6 = 011000$,
- $280 \cdot 61 = 70 = 2 + 3 + 13 + 52 = 110101$,
- $333 \cdot 61 = 48 = 2 + 6 + 13 + 27 = 101110$.

The final decoding is done using the simple easy knapsack,

$$\{2, 3, 6, 13, 27, 52\},$$

and our earlier linear time algorithm for super-increasing knapsack problems.

Our example knapsack problem of six items is too small; typically one would have at least 250 items. The values of N and M should also be around 400 bits. But, even with parameters as large as these, the above Merkle–Hellman scheme has been broken using lattice based techniques, using a method which we will now explain.

If $\{w_1, \ldots, w_n\}$ are a set of knapsack weights then we define the density of the knapsack to be

$$d = \frac{n}{\max\{\log_2 w_i : 1 \le i \le n\}}.$$

One can show, using the following method, that a knapsack with low density will be easy to solve using lattice basis reduction. Why this allows us to break the Merkle–Hellman scheme is that the Merkle–Hellman construction will always produce a low-density public knapsack.

Suppose we wish to solve the knapsack problem given by the weights $\{w_1, \ldots, w_n\}$ and the target S. Consider the lattice L of dimension $n + 1$ generated by columns of the following matrix:

$$A = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 & \frac{1}{2} \\ 0 & 1 & 0 & \ldots & 0 & \frac{1}{2} \\ 0 & 0 & 1 & \ldots & 0 & \frac{1}{2} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & \frac{1}{2} \\ w_1 & w_2 & w_3 & \ldots & w_n & S \end{pmatrix}.$$

Now, since we are assuming there is a solution to our knapsack problem, given by the bit vector $(b_1, \ldots, b_n)$, we know that the vector

$$y = A \cdot x,$$

is in our lattice, where $x = (b_1, \ldots, b_n, -1)$. But the components of y are given by

$$y_i = \begin{cases} b_i - \frac{1}{2} & 1 \le i \le n \\ 0 & i = n + 1. \end{cases}$$

Hence, the vector y is very short since it has length bounded by

$$\|y\| = \sqrt{{y_1}^2 + \cdots + {y_{n+1}}^2} < \frac{\sqrt{n}}{2}.$$

But a low-density knapsack will usually result in a lattice with relatively large discriminant, hence the vector y is exceptionally short in the lattice. If we now apply the LLL algorithm to the matrix A we obtain a new basis matrix A'. The first basis vector a'_1 of this LLL reduced basis is then likely to be the smallest vector in the lattice and so we are likely to have

$$a'_1 = y.$$

But given y we can then solve for x and recover the solution to the original knapsack problem.

As an example we take out earlier knapsack problem of

$$b_1 62 + b_2 93 + b_3 81 + b_4 88 + b_5 102 + b_6 37 = 174.$$

We form the matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 1 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 1 & \frac{1}{2} \\ 62 & 93 & 81 & 88 & 102 & 37 & 174 \end{pmatrix}.$$

We apply the LLL algorithm to this matrix so as to obtain the new lattice basis,

$$A' = \frac{1}{2}\begin{pmatrix} 1 & -1 & -2 & 2 & 3 & 2 & 0 \\ -1 & -3 & 0 & -2 & -1 & -2 & 0 \\ -1 & -1 & -2 & 2 & -1 & 2 & 0 \\ 1 & -1 & -2 & 0 & -1 & -2 & -2 \\ 1 & -1 & 0 & 2 & -3 & 0 & 4 \\ 1 & 1 & 0 & -2 & 1 & 2 & 0 \\ 0 & 0 & -2 & 0 & 0 & -2 & 2 \end{pmatrix}.$$

We write

$$y = \frac{1}{2}\begin{pmatrix} 1 \\ -1 \\ -1 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix},$$

and compute

$$x = A^{-1} \cdot y = \begin{pmatrix} 0 \\ -1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

So we see that we take

$$(b_1, b_2, b_3, b_4, b_5, b_6) = (0, 1, 1, 0, 0, 0),$$

as a solution to our knapsack problem.

3. Bit Security

Earlier we looked at decision problems, i.e. those problems which output a single bit, and showed that certain other non-decision problems, such as the knapsack problem, could be reduced to looking at decision problems only. A similar situation arises in cryptography where we wish to know whether computing a single bit of information is as hard as computing all of the information.

For example suppose one is using the RSA function

$$x \longmapsto y = x^e \pmod{N}.$$

It may be that in a certain system the attacker only cares about computing $b = x \pmod 2$ and not the whole of x. We would like it to be true that computing even this single bit of information about x is as hard as computing all of x. In other words we wish to study the so-called bit security of the RSA function.

We can immediately see that bit security is related to semantic security. For example if an attacker could determine the parity of an underlying plaintext given only the ciphertext they could easily break the semantic security of the encryption algorithm.

First we define some notation:

DEFINITION 19.5. *Let $f : S \to T$ be a one-way function where S and T are finite sets and let $B : S \to \{0, 1\}$ denote a binary function (called a predicate). A hard predicate $B(x)$ for f is one which is easy to compute given $x \in S$ and for which it is hard to compute $B(x)$ given only $f(x)$.*

The way one proves a predicate is a hard predicate, assuming f is a one-way function, is to assume we are given an oracle which computes $B(x)$ given $f(x)$, and then show that this oracle can be used to easily invert f.

A k-bit predicate and hard k-bit predicate are defined in an analogous way but now assuming the codomain of B is the set of bit strings of length k rather than just single bits. We would like to show that various predicates, for given cryptographically useful functions f, are in fact hard predicates.

3.1. Hard Predicates for Discrete Logarithms. Let G denote a finite abelian group of prime order q and let g be a generator. Consider the predicate

$$B_2 : x \longmapsto x \pmod 2$$

we can show

THEOREM 19.6. *The predicate B_2 is a hard predicate for the function*

$$x \longmapsto g^x.$$

PROOF. Let $\mathcal{O}(h, g)$ denote an oracle which returns the least significant bit of the discrete logarithm of h to the base g, i.e. it computes $B_2(x)$ for $x = \log_g h$. We need to show how to use $\mathcal{O}$ to solve a discrete logarithm problem.

Suppose we are given $h = g^x$, we perform the following steps. First we let $t = \frac{1}{2} \pmod q$, then we set $y = 0$, $z = 1$ and compute until $h = 1$ the following steps:

- $b = \mathcal{O}(h, g)$.
- If $b = 1$ then $y = y + z$ and $h = h/g$.
- Set $h = h^t$ and $z = 2 \cdot z$.

We then output y as the discrete logarithm of h with respect to g. □

To see this work consider the field $\mathbb{F}_{607}$ and the element $g = 64$ of order $q = 101$. We wish to find the discrete logarithm of $h = 56$ with respect to g. Using the algorithm in the above proof we compute

h	$\mathcal{O}(h,g)$	z	y
56	0	1	0
451	1	2	2
201	1	4	6
288	0	8	6
100	1	16	22
454	0	32	22
64	1	64	86

One can indeed check that

$$g^{86} = h \pmod{p}.$$

3.2. Hard Predicates for the RSA Problem. The RSA problem, namely given $c = m^e \pmod{N}$ has the following three hard predicates:

- $B_1(m) = m \pmod 2$.
- $B_h(m) = 0$ if $m < N/2$ otherwise $B_h(m) = 1$.
- $B_k(m) = m \pmod{2^k}$ where $k = O(\log\log N)$.

We denote the corresponding oracles by $\mathcal{O}_1(c,N)$, $\mathcal{O}_h(c,N)$ and $\mathcal{O}_k(c,N)$. We do not deal with the last of these but we note that the first two are related since,

$$\mathcal{O}_h(c,N) = \mathcal{O}_1(c\cdot 2^e \pmod N, N),$$
$$\mathcal{O}_1(c,N) = \mathcal{O}_h(c\cdot 2^{-e} \pmod N, N).$$

We then have, given an oracle for $\mathcal{O}_h$ or $\mathcal{O}_1$, that we can invert the RSA function using the following algorithm, which is based on the standard binary search algorithm. We let $y = c$, $l = 0$ and $h = N$, then while $h - l \geq 1$ we perform

- $b = \mathcal{O}_h(y,N)$,
- $y = y\cdot 2^e \pmod N$,
- $m = (h+l)/2$,
- If $b = 1$ then set $l = m$, otherwise set $h = m$.

On exiting the above loop the value of $\lfloor h \rfloor$ should be the preimage of c under the RSA function.

As an example suppose we have

$$N = 10\,403 \text{ and } e = 7$$

as the public information and we wish to invert the RSA function for the ciphertext $c = 3$ using the oracle $\mathcal{O}_h(y,N)$

y	$\mathcal{O}(y, N)$	l	h
3	0	0	10 403
$3 \cdot 2^7$	1	0	5201.5
$3 \cdot 4^7$	1	2600.7	5201.5
$3 \cdot 8^7$	1	3901.1	5201.5
$3 \cdot 16^7$	0	4551.3	5201.5
$3 \cdot 32^7$	0	4551.3	4876.4
$3 \cdot 64^7$	1	4551.3	4713.8
$3 \cdot 128^7$	0	4632.5	4713.8
$3 \cdot 256^7$	1	4632.5	4673.2
$3 \cdot 512^7$	1	4652.9	4673.2
$3 \cdot 1024^7$	1	4663.0	4673.2
$3 \cdot 2048^7$	1	4668.1	4673.2
$3 \cdot 4096^7$	1	4670.7	4673.2
$3 \cdot 8192^7$	0	4671.9	4673.2
-	-	4671.9	4672.5

So the preimage of 3 under the RSA function

$$x \longmapsto x^7 \pmod{10\,403}$$

is 4672.

4. Random Self-reductions

We remarked earlier, when considering the Merkle–Hellman scheme and other schemes based on complexity theory, that the problem with these schemes is that the associated problems were hard in the worst case but easy on average. The obvious question to ask oneself is how does one know that problems such as the RSA problem or the DDH problem also do not have this property? For example given an RSA modulus N and a public exponent e it might be hard to solve

$$c = m^e \pmod{N}$$

for a random c in the worst case, but it could be easy on average.

It turns out that one can prove that problems such as RSA for a fixed modulus N or DDH for a fixed group G are hard on average. The technique to do this is based on a random self-reduction from one given problem instance to another random problem instance. This means that if we can solve the problem on average then we can solve the problem in the worst case. Hence, the worst case behaviour of the problem and the average case behaviour of the problem must be similar.

LEMMA 19.7. *The RSA problem is random self-reducible.*

PROOF. Suppose we are given c and are asked to solve

$$c = m^e \pmod{N},$$

where the idea is that this is a 'hard' problem instance. We reduce this to an 'average' problem instance by choosing $s \in (\mathbb{Z}/N\mathbb{Z})^*$ at random and setting

$$c' = s^e c.$$

We then try to solve

$$c' = m'^e \pmod{N}.$$

If we are unsuccessful we choose another value of s until we hit the 'average' type problem. If the average case was easy then we could solve $c' = m'^e \pmod{N}$ for m' and then set

$$m = \frac{m'}{s}$$

and terminate. □

One can also show that the DDH problem is random self-reducible, in the sense that testing whether

$$(x, y, z) = (g^a, g^b, g^c)$$

is a valid Diffie–Hellman triple, i.e. whether $c = a \cdot b$, does not depend on the particular choices of a, b and c. To see this consider the related triple

$$(x', y', z') = (g^{a_1}, g^{b_1}, g^{c_1}) = (x^v g^{u_1}, y g^{u_2}, z^v y^{u_1} x^{v u_2} g^{u_1 u_2})$$

for random u_1, u_2, v. Now if (x, y, z) is a valid Diffie–Hellman triplet then so is (x', y', z'), and vice versa.

One can show that the distribution of (x', y', z') will be uniform over all valid Diffie–Hellman triples if the original triple is a valid Diffie–Hellman triple, whilst the distribution will be uniform over all triples (and not just Diffie–Hellman ones) in the case where the original triple was not a valid Diffie–Hellman triple.

5. Randomized Algorithms

We end this chapter with a discussion of randomized algorithms; first we give some definitions of algorithm types, then we relate these to complexity classes and finally we give some examples.

Recall that definitions are usually given for decision problems, so assume we have a property which we wish to test to be true or false. There are the following definitions of algorithm types, all taking their names from gambling cities.

- **Monte-Carlo algorithm**
  - Always outputs **false** if the answer is actually **false**
  - Answers **true** with probability $\geq 1/2$.
  - Otherwise answers **false**, even though the actual answer is **true**.
- **Atlantic City algorithm**
  - Outputs **true** with probability $\geq 2/3$ of being correct.
  - Outputs **false** with probability $\geq 2/3$ of being correct.
- **Las Vegas algorithm**
  - Will terminate with the correct answer with probability $\geq 1/2$.
  - Otherwise will not terminate.

In the above definitions we assume that the algorithm runs in polynomial time in terms of the size of the input data. We can clearly extend these definitions to non-decision problems quite easily.

We now turn our attention to the randomized complexity classes. We assume we have a problem instance I and a possible witness w whose length is a polynomial function of the length of the instance I. We wish to determine whether $I \in S$.

DEFINITION 19.8. *A problem $\mathcal{D}P$ is said to lie in class $\mathcal{R}P$ if there is an algorithm A, which on input of a problem instance I and a witness w will perform as follows:*

- *If $I \in S$ then for at least half of all possible witnesses w the algorithm A outputs that $I \in S$.*
- *If $I \notin S$ then for all witnesses w the algorithm A outputs $I \notin S$.*

Note that we have, since we can replace 'at least half' with 'at least one', that

$$\mathcal{R}P \subset \mathcal{N}P.$$

We also clearly have $\mathcal{P} \subset \mathcal{R}P$.

The class $\mathcal{RP}$ is important since it gives us a probabilistic algorithm to decide whether $I \in S$ for any value of S. We generate k random witnesses w_i and call $A(I, w_i)$ for $i = 1, \ldots, k$. Then if $A(I, w_i)$ returns $I \in S$ for at least one value of w_i then we return $I \in S$, otherwise we return $I \notin S$. This latter statement will then be false around $1/2^k$ of the time. Relating this to our definitions based on gambling cities we see that

$$\text{If } \mathcal{DP} \in \mathcal{RP} \text{ there exists a Monte-Carlo algorithm for } \mathcal{DP}.$$

We now define another class.

DEFINITION 19.9. *A problem $\mathcal{DP}$ is said to lie in class $\mathcal{BPP}$ if there is an algorithm A, which on input of a problem instance I and a witness w will perform as follows:*

- *If $I \in S$ then for at least $2/3$ of all possible witnesses w the algorithm A outputs $I \in S$.*
- *If $I \notin S$ then for at least $2/3$ of all possible witnesses w the algorithm A outputs $I \notin S$.*

We see that

$$\text{If } \mathcal{DP} \in \mathcal{BPP} \text{ there exists an Atlantic City algorithm for } \mathcal{DP}.$$

Finally we define the class $\mathcal{ZPP}$ as

$$\mathcal{ZPP} = \mathcal{RP} \cap \text{co} - \mathcal{RP},$$

with

$$\text{If } \mathcal{DP} \in \mathcal{ZPP} \text{ there exists a Las Vegas algorithm for } \mathcal{DP}.$$

We have the inclusions

$$\mathcal{P} \subset \mathcal{ZPP} \subset \mathcal{RP} \subset \mathcal{NP} \cap \mathcal{BPP}.$$

The Fermat test and the Miller–Rabin test for primality are examples of Monte-Carlo algorithms to test for compositeness. Recall these tests on input of a number N do one of two things:

- If N is prime will always output **false** (or probably prime).
- If N is composite will output **true** with probability $\geq 1/2$, otherwise will output **false**.

Hence

$$\texttt{COMPOSITES} \in \mathcal{RP}.$$

By repeating the Monte-Carlo test we can amplify the probability of being correct to be arbitrarily close to one.

The Adleman–Huang algorithm for primality proving is an example of a Las Vegas algorithm to test for primality. The input to this problem is a number N. If the input is composite then the algorithm may not terminate, but if it does terminate then it will tell us correctly whether N is prime or not. Hence

$$\texttt{PRIMALITY} \in \mathcal{ZPP}.$$

The historically earlier ECPP algorithm, on which the test of Adleman and Huang is based, is not guaranteed to terminate on input of a prime number, but in practice it always does.

Chapter Summary

- Complexity theory deals with the worst case behaviour of algorithms to solve a given decision problem.

- Some problems are easy on average, but there exist certain instances which are very hard to solve. We do not wish to base our cryptographic systems on such problems.
- Cryptographic systems based on knapsack problems have been particularly notorious from this perspective, as one can often use lattice basis reduction to break them.
- Problems which we do base public key cryptography on, such as the RSA problem or the discrete logarithm problem, have the property that even computing single bits of the answer seems to be as hard as computing the whole answer.
- Problems such as RSA and the DDH problem are hard on average, since they posses random self-reductions from a given instance of the problem to a random instance of the problem.

Further Reading

A nice introduction to complexity theory can be found in Chapter 2 of Bach and Shallit. A discussion of the relationships between theoretical complexity theory and cryptographic concepts such as zero-knowledge proofs can be found in the book by Goldreich. A discussion of knapsack based systems and how to break them using lattices can be found in the survey article by Odlyzko.

E. Bach and S. Shallit. *Algorithmic Number Theory, Volume 1: Efficient Algorithms.* MIT Press, 1996.

O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudo-randomness.* Springer-Verlag, 1999.

A. Odlyzko. *The Rise and Fall of Knapsack Cryptosystems.* In Cryptology and Computational Number Theory, Proc. Symposia in Applied Maths, Volume 42, 1990.

CHAPTER 20

Provable Security: With Random Oracles

Chapter Goals

- To describe the random oracle model.
- To show how the random oracle model can be used to prove certain signature schemes are secure.
- To show why chosen ciphertext attacks require our public key encryption algorithms to have redundancy in the ciphertext.
- To explain RSA-OAEP and give a proof sketch.
- To describe how to turn ElGamal encryption into a secure system.

1. Introduction

The modern approach to showing that certain protocols are secure is one based on provable security. This name is in fact a misnomer since the techniques used do not actually prove security, in the sense of perfect security mentioned earlier. Instead, the proponents of provable security aim to prove that if an adversary is able to break a certain notion of the security of a system then one could use the adversary to do something believed to be impossible.

For example one tries to prove that if one can break the semantic security of RSA in a chosen ciphertext attack then one is also able to factor integers. Hence, such a proof is a relativized result, it is a proof of security relative to the hardness of factoring integers.

The major contribution of the area of provable security has really been in actually defining what is meant by a secure encryption or signature algorithm. Many of the concepts we have already introduced such as existential forgery, semantic security, indistinguishability of encryptions, adaptive chosen ciphertext attacks have all arisen due to the study of provable security.

Let us explain the techniques of provable security by explaining what one means in a concrete example. We suppose we are given an adversary which is a probabilistic algorithm which breaks some security property of RSA (for example semantic security of RSA) with a certain non-negligible probability. Already this leads us to some definitional work, what do we mean by non-negligible probability?

We hence need to define what we mean much better. We now assume that the scheme has a security parameter k, which measures how big the key size is. For example in RSA the security parameter could be the number of bits in the modulus N. The adversary is said to be successful with non-negligible probability if it succeeds in its task with probability greater than

$$1/p(k),$$

where p is some polynomial in k.

For the moment let us assume that our adversary A is a passive adversary, i.e. for RSA encryption it makes no decryption queries. We now wish to present a new algorithm B_A which takes as input an integer N and which calls the adversary a polynomial, in k, number of times. This new

algorithm's aim could be to output the factors of N, with again a non-negligible probability. The algorithm B_A would show that the existence of such an adversary A would imply a polynomial-time factoring algorithm, which succeeded with a non-negligible probability. Since we do not believe a polynomial-time factoring algorithm is achievable with current knowledge we can conclude that such an adversary is also unachievable with current knowledge.

You have already seen the above technique used when we showed that a successful passive adversary against the ElGamal encryption scheme would imply an efficient algorithm to solve the DDH problem, or when we showed that a passive adversary against the Goldwasser–Micali scheme would imply an efficient algorithm to solve the QUADRES problem.

To recap, given an algorithm A we create a new algorithm B_A which uses A as a subroutine. The input to B_A is the hard mathematical problem we wish to solve, whilst the input to A is some cryptographic problem. The difficulty arises when A is an active adversary, in this case A is allowed to call a decryption or signature oracle for the input public keys. The algorithm B_A, if it wants to use algorithm A as a subroutine, needs to supply the answers to A's oracle queries. Algorithm B_A now has a number of problems:

- Its responses must appear valid, in that encryptions should decrypt and signatures should verify, otherwise algorithm A would notice its oracle was lying. Hence, algorithm B_A could no longer guarantee that algorithm A was successful with non-negligible probability.
- The responses of the oracle should be consistent with the probability distributions of responses that A expects if the oracles were true decryption/encryption oracles. Again, otherwise A would notice.
- The responses of the oracles should be consistent across all the calls made by the adversary A.
- Algorithm B_A needs to supply these answers without knowing the secret key. For example in the case of RSA, if B_A wants to find the factors of N, it can hardly use these factors to respond to algorithm A before it has found the factors.

This last point is the most crucial one. We are essentially asking B_A to decrypt or sign a message without knowing the private key, but this is meant to be impossible since our scheme is meant to be secure.

To get around this problem it has become common practice to use something called the 'random oracle model'. A random oracle is an idealized hash function which on input of a new query will pick, uniformly at random, some response from its output domain, and which if asked the same query twice will always return the same response.

In the random oracle model we assume our adversary A makes no use of the explicit hash function defined in the scheme under attack. In other words the adversary A runs, and is successful, even if we replace the real hash function by a random oracle. The algorithm B_A responds to the decryption oracle and/or signature queries of A by cheating and 'cooking' the responses of the random oracle to suit his own needs. To see how this is done in practice look at the next section on proofs of security of signature algorithms.

A proof in the random oracle model is an even more relativized proof than that which we considered before. Such a proof says that assuming some problem is hard, say factoring, then an adversary cannot exist which makes no use of the underlying hash function. This does not imply that an adversary does not exist which uses the real specific hash function as a means of breaking the cryptographic system.

In all our proofs and definitions we are very loose. We try to convey the flavour of the arguments rather than the precise details. Those who want more precise definitions should look at the original papers. One should be warned however that in this field definitions can alter quite subtly from one

paper to the next, this is because most of the real importance is in the definitions rather than the proofs themselves.

2. Security of Signature Algorithms

We first consider proofs of security of digital signature algorithms because they are conceptually somewhat simpler.

The first main technique we shall introduce is the forking lemma, due to Stern and Pointcheval. This applies to certain types of signature schemes which use a hash function as follows: To sign a message

- the signer produces a, possibly empty, commitment σ_1,
- the signer computes $h = H(\sigma_1 \| m)$,
- the signer computes σ_2 which is the 'signature' on σ_1 and h.

We label the output of the signature schemes as $(\sigma_1, H(\sigma_1\|m), \sigma_2)$ so as to keep track of the exact hash query. For example we have

- DSA : $\sigma_1 = \emptyset$, $h = H(m)$,
$$\sigma_2 = (r, (h + xr)/k \pmod q),$$
where $r = (g^k \pmod p) \pmod q$.
- EC-DSA : $\sigma_1 = \emptyset$, $h = H(m)$,
$$\sigma_2 = (r, (h + xr)/k \pmod q),$$
where $r = x\text{-coord}([k]G)$.
- Schnorr signatures: $\sigma_1 = g^k$, $h = H(\sigma_1\|m)$
$$\sigma_2 = xh + k \pmod q.$$

In all of these schemes the hash function is assumed to have codomain equal to $\mathbb{F}_q$.

Recall in the random oracle model the hash function is allowed to be cooked up by the algorithm B_A to do whatever it likes. Suppose an adversary A can produce an existential forgery on a message m with non-negligible probability in the random oracle model. Hence, the output of the adversary is
$$(m, \sigma_1, h, \sigma_2).$$
We can assume that the adversary makes the critical hash query
$$h = H(\sigma_1\|m),$$
since otherwise we can make the query for the adversary ourselves.

Algorithm B_A now runs the adversary A twice, with the same random tape and a slightly different random oracle. The adversary A runs in polynomial time and so makes polynomially many hash queries. If all hash queries were answered the same as before then algorithm A would output exactly the same signature. However, algorithm B_A answers these random oracle queries just as before, but chooses one hash query at random to answer differently. With non-negligible probability this will be the critical hash query and so (with non-negligible probability) the adversary B_A will obtain two signatures on the same message which have different hash query responses. In other words we obtain
$$(m, \sigma_1, h, \sigma_2) \text{ and } (m, \sigma_1, h', \sigma_2').$$
We then try to use these two outputs of algorithm A to solve the hard problem which is the goal of algorithm B_A. The exact details and a thorough proof of this technique can be found in the paper by Stern and Pointcheval mentioned in the Further Reading section at the end of this chapter.

2.1. Passive Adversary Examples.

We concern ourselves with some applications.

2.1.1. *Schnorr Signatures:* The technique of the forking lemma allows us to show

THEOREM 20.1. *In the random oracle model, assuming discrete logarithms are hard to compute for the group G, no passive adversary against Schnorr signatures can exist for the group G.*

PROOF. Let the input to algorithm B_A be a discrete logarithm problem $y = g^x$ which we wish to solve. So let us assume we run our adversary A on input of the public key y and try to use the forking lemma argument. With non-negligible probability we obtain two signatures

$$\left(m, \sigma_1 = g^k, h, \sigma_2 = xh + k \pmod{q}\right)$$

and

$$\left(m, \sigma_1' = g^{k'}, h', \sigma_2' = xh' + k' \pmod{q}\right),$$

where $h = H(\sigma_1 \| m)$ is the oracle query from the first run of A and $h' = H(\sigma_1' \| m)$ is the oracle query from the second run of A.

The algorithm B_A's goal is to recover x. It concludes that we must have $k = k'$ since we have $\sigma_1 = \sigma_1'$. So it concludes that

$$Ax = B \pmod{q},$$

where

$$A = h - h' \pmod{q},$$
$$B = \sigma_2 - \sigma_2' \pmod{q}.$$

It will also know that $A \neq 0$, since otherwise the two hash values would be equal. This then means that B_A can solve the required discrete logarithm problem by computing

$$x = A^{-1}B \pmod{q}.$$

□

We shall later show that Schnorr signatures are also secure against active adversaries in the random oracle model.

2.1.2. *DSA Signatures:* The above argument for Schnorr signatures does not apply to DSA as we shall now show.

Let the input to algorithm B_A be a discrete logarithm problem $y = g^x$ which we wish to solve. So let us assume we run our adversary A on input of the public key y and try to use the forking lemma argument. With non-negligible probability we obtain two signatures

$$(m, \sigma_1 = \emptyset, h, \sigma_2 = (r, s)) \text{ and } (m, \sigma_1' = \emptyset, h', \sigma_2' = (r', s')),$$

where $h = H(m)$ is the oracle query from the first run of A, $h' = H(m)$ is the oracle query from the second run of A and

$$r = g^k \pmod{p} \pmod{q},$$
$$r' = g^{k'} \pmod{p} \pmod{q},$$
$$s = (h + xr)/k \pmod{q},$$
$$s' = (h' + xr')/k' \pmod{q}.$$

The algorithm B_A's goal is to recover x. We can no longer conclude that $k = k'$, since we do not even know that $r = r'$. Hence, the proof technique does not apply. In fact there is no known proof of security for DSA signatures.

We can try and repair this by using a modified form of DSA by applying the hash function to m and r instead of just m. In the context of the notation used in our description of the forking lemma this would imply that $\sigma_1 = r$, $h = H(m \| r)$ and $\sigma_2 = (h + xr)/k \pmod{q}$.

Even with this modification, to make DSA more like Schnorr signatures, we cannot prove security. Our forking lemma argument would imply that we obtain two signatures with

$$\begin{aligned} r &= g^k \pmod p \pmod q, \\ r' &= g^{k'} \pmod p \pmod q, \\ s &= (h + xr)/k \pmod q, \\ s' &= (h' + xr')/k' \pmod q, \end{aligned}$$

and with $r = r'$. But we still could not imply that $k = k'$ since this does not follow from the equation

$$g^k \pmod p \pmod q = g^{k'} \pmod p \pmod q.$$

It is the reduction modulo q which is getting in the way, if we removed this then our signature scheme could be shown to be secure. But we would lose the property of small signature size which DSA has.

2.1.3. *EC-DSA Signatures:* A similar problem arises with EC-DSA and the forking lemma argument. But now a small modification allows us to give a proof of security for the modified scheme, in certain situations. We again assume that we apply the hash function to m and r instead of just m. We then obtain by running our adversary twice and using the forking lemma

$$\begin{aligned} r &= x\text{-coord}([k]P) \pmod q, \\ r' &= x\text{-coord}([k']P) \pmod q, \\ s &= (h + xr)/k \pmod q, \\ s' &= (h' + xr')/k' \pmod q, \end{aligned}$$

where again we have $r = r'$ and $h = H(m\|r)$ and $h' = H(m\|r')$ are the two critical hash queries. If q is larger than the size of the finite field, from $r = r'$ we can now conclude that $k = \pm k'$ and so we deduce that

$$(s \mp s')k = h - h' \pmod q.$$

We therefore obtain two possibilities for k and we recover two possibilities for x. The actual answer is produced by checking which one of our two possibilities for x satisfies $[x]P = Y$. So we have shown:

THEOREM 20.2. *In the random oracle model the above* **modified** *version of EC-DSA is secure against passive adversaries, assuming that the discrete logarithm problem in $E(\mathbb{F}_p)$ is hard and*

$$q = \#E(\mathbb{F}_p) > p.$$

Notice how this result only applies to a certain subset of all elliptic curves.

2.2. Active Adversaries. To provide a proof for active adversaries we need to show how the algorithm B_A will answer the signature queries of the algorithm A. To do this we use the random oracle model again, in that we use the ability of B_A to choose the output of the hash function. Notice that this may mean that the input to the hash function must be unknown up until the message is signed. If the hash function is only applied to the message m and not some other quantity (such as σ_1 above) then algorithm A could have already queried the hash oracle for the input m before the signature is created, and then algorithm B_A would not be able to change the response from the previous one.

The process of B_A signing signatures for A without A being able to tell and without B_A having the private key is called a simulation of the signing queries. This simulation essentially means that an active attack can be no more powerful than a passive attack, in the random oracle model, since any active attacker can be turned into a passive attacker by simply simulating the signing queries.

2.2.1. *Schnorr Signatures:* Providing a simulation of the signing queries for Schnorr signatures is particularly easy, and the simulator is closely related to our zero-knowledge protocols of Chapter 25. We assume that the simulator keeps a list L of all previous random oracle queries. On input of a message m the simulator does the following:

(1) Computes random values of s and h such that $1 \leq s, h < q$.
(2) Sets $r = g^s y^{-h}$.
(3) If $(r\|m, h') \in L$ for $h' \neq h$ then the simulator returns to step 1.
(4) Sets $L = L \cup (r\|m, h)$, i.e. the hash oracle should now always return h on input of $(r\|m)$.
(5) Output the 'signature' (h, s).

You should check that the above does produce a valid signature, and that assuming h is a random oracle the above simulation cannot be distinguished by algorithm A from a true signature algorithm. Hence we have:

THEOREM 20.3. *In the random oracle model, assuming discrete logarithms are hard to compute for the group G, no active adversary against Schnorr signatures can exist for the group G.*

No similar security statement is known for DSA. A security proof is known for EC-DSA where, instead of modelling the hash function as a generic object and reducing the security to a discrete logarithm problem, one models the group operation as a generic object and reduces the security of EC-DSA to the collision resistance of the actual hash function used.

2.3. RSA with Full Domain Hash. One should notice that in the previous discussion of the Schnorr, DSA and EC-DSA signature schemes we assumed that the hash function H is a function with codomain $\mathbb{F}_q$. Such hash functions are hard to construct in practice, but the above arguments assume this property.

A similar situation occurs with a variant of RSA signatures called RSA-FDH, or *full domain hash*. In this we assume a hash function

$$H : \{0,1\}^* \longrightarrow (\mathbb{Z}/N\mathbb{Z})^*,$$

where N is the RSA modulus. Again such hash functions are hard to construct in practice, but if we assume they can exist and we model them by a random oracle then we can prove the following RSA signature algorithm is secure.

Let N denote an RSA modulus with public exponent e and private exponent d. Let f denote the function

$$f : \begin{cases} (\mathbb{Z}/N\mathbb{Z})^* \longrightarrow (\mathbb{Z}/N\mathbb{Z})^* \\ x \longmapsto x^e. \end{cases}$$

The RSA problem is given $y = f(x)$ determine x. In the RSA-FDH signature algorithm on being given a message m we produce the signature

$$s = H(m)^d = f^{-1}(H(m)).$$

One can then prove the following theorem.

THEOREM 20.4. *In the random oracle model if an active adversary A exists which produces an existential forgery for RSA-FDH, which requires q_H hash queries and q_S signature queries, then there is an algorithm which given y can invert the RSA function on y with probability $1/q_H$.*

PROOF. We describe an algorithm B_A which on input of $y \in (\mathbb{Z}/N\mathbb{Z})^*$ outputs $x = f^{-1}(y)$. Algorithm B_A first chooses a value of $t \in [1, \ldots, q_H]$ and throughout keeps a numbered record of all the hash queries made. Algorithm B_A runs algorithm A and responds to the hash queries for the input m_i as follows:

- If A makes a hash query, and this is the tth such query then B_A replies with y and updates the internal hash list so that $y = H(m_t)$.

- If A makes a hash query m_i with $i \neq t$, then B_A computes a random $s_i \in (\mathbb{Z}/N\mathbb{Z})^*$ and updates the internal hash list so that $H(m_i) = s_i^e \pmod N = h_i$, keeping a record of the value of s_i. Algorithm B_A then responds with h_i.

If A makes a signing query for a message m_i before making a hash query on the message m_i then B_A first makes the hash query for algorithm A. Then signing queries are responded to as:

- If message m_i is equal to m_t then algorithm B_A stops and returns fail.
- If $m_i \neq m_t$ then B_A returns s_i as a response to the signature query.

Let A terminate with output (m, s) and without loss of generality we can assume that A made a hash oracle query for the message m. Now if $m \neq m_t$ then B_A terminates and admits failure. But if $m = m_t$ then we have

$$f(s) = H(m_t) = y.$$

Hence we have succeeded in inverting f.

In analysing algorithm B_A one notices that if A terminates successfully then (m, s) is an existential forgery and so m was not asked of the signing oracle. The value of t is independent of the view of A, so A cannot try and always ask for the signature of message m_t in the algorithm rather than not ask for the signature. Hence, roughly speaking, the probability of success is around $1/q_H$, i.e. the probability that the existential forgery was on the message m_t and not on some other one. □

2.4. RSA-PSS. Another way of securely using RSA as a signature algorithm is to use a system called RSA-PSS, or *probabilistic signature scheme*. This scheme can also be proved secure in the random oracle model under the RSA assumption. We do not give the details of the proof here but simply explain the scheme, since it is becoming increasingly important due to its adoption by standards bodies. The advantage of RSA-PSS over RSA-FDH is that one only requires a hash function with a traditional codomain, e.g. bit strings of length t, rather than a set of integers modulo another number.

As usual one takes an RSA modulus N, a public exponent e and a private exponent d. Suppose the security parameter is k, i.e. N is a k bit number. We define two integers k_0 and k_1 so that

$$k_0 + k_1 \leq k - 1.$$

For example one could take $k_i = 128$ or 160.

We then define two hash functions, one which expands data and one which compresses data:

$$G : \{0,1\}^{k_1} \longrightarrow \{0,1\}^{k-k_1-1}$$
$$H : \{0,1\}^* \longrightarrow \{0,1\}^{k_1}.$$

We let

$$G_1 : \{0,1\}^{k_1} \longrightarrow \{0,1\}^{k_0}$$

denote the function which returns the first k_0 bits of $G(w)$ for $w \in \{0,1\}^{k_1}$ and we let

$$G_2 : \{0,1\}^{k_1} \longrightarrow \{0,1\}^{k-k_0-k_1-1}$$

denote the function which returns the last $k - k_0 - k_1 - 1$ bits of $G(w)$ for $w \in \{0,1\}^{k_1}$. To sign a message m:

- Generate a random value $r \in \{0,1\}^{k_0}$.
- Put $w = H(m\|r)$.
- Set $y = 0\|w\|(G_1(w) \oplus r)\|G_2(w)$.
- Output $s = y^d \pmod N$.

To verify a signature (s, m):

- Compute $y = s^e \pmod N$.

- Split y into the components
$$b\|w\|\alpha\|\gamma$$
where b is one bit long, w is k_1 bits long, α is k_0 bits long and γ is $k - k_0 - k_1 - 1$ bits long.
- Compute $r = \alpha \oplus G_1(w)$.
- The signature verifies if
$$b = 0 \text{ and } G_2(w) = \gamma \text{ and } H(m\|r) = w.$$

If we allow the modelling of the hash functions G and H by random oracles then one can show that the above signature algorithm is secure, in the sense that the existence of a successful algorithm to find existential forgeries could be used to produce an algorithm to invert the RSA function. For the proof of this one should consult the EuroCrypt '96 paper of Bellare and Rogaway mentioned in the Further Reading section at the end of this chapter.

3. Security of Encryption Algorithms

We have seen that it is easy, under the DDH assumption, to produce semantically secure public key encryption schemes assuming only passive adversaries. For example the ElGamal encryption scheme satisfies these properties. We have also seen that a semantically secure system based on the QUADRES problem is easy to produce, assuming only passive adversaries, but this system of Goldwasser and Micali has terrible message expansion properties. It is much harder to produce discrete logarithm based systems which are secure against active adversaries or which are semantically secure under the RSA assumption.

In this section we first present some historical attempts at producing ElGamal based encryption algorithms which aimed to be secure against active adversaries. These are important as they show a basic design criteria. We then go on to describe the main RSA based system which is secure against active adversaries in the random oracle model, namely RSA-OAEP.

3.1. Immunization of ElGamal Based Encryption. Recall that ElGamal encryption is given by
$$(g^k, m \cdot y^k)$$
where $y = g^x$ is the public key. Such a system can be proved to have semantic security under the DDH assumption using quite elementary techniques, as we showed in Chapter 18. However, we also showed that such a system is not secure against active adversaries since the ciphertext was trivially malleable.

It was soon realized that the problem with active attacks was that it was too easy for the adversary to write down a valid ciphertext, and not just a related one. The reasoning went that if it was hard for the adversary to write down a valid ciphertext without having first encrypted the plaintext to produce the ciphertext, then the adversary would have no advantage in mounting a chosen ciphertext attack. After all, why would an adversary want to decrypt a ciphertext if the only way he could produce a ciphertext was to encrypt some plaintext?

This meant one needed a decryption function which on input of a ciphertext would either output the corresponding plaintext or would output the $\perp$ symbol, to signal an Invalid Ciphertext. For this to happen some redundancy needs to be added to the ciphertext which could be checked by the decryptor, to check whether the ciphertext was valid. Compare this with our discussion on encryption functions in Chapter 5, where we argued that a ciphertext should contain no redundancy. But there we were only interested in passive attacks, here we are trying to defend against much more powerful adversaries.

Zheng and Seberry were the first to explore this philosophy for practical cryptosystems, which pervades the modern approach to public key encryption function design. Their overall approach is

important, so we present these early attempts at producing secure public key encryption schemes as illustrative of the approach.

The first thing to notice is that public key encryption is usually used to convey a key to encrypt a large message, hence it is not necessary to encrypt a message which lies in the group G (as in ElGamal encryption). We can still use the ElGamal idea to transport a key, which we can then use to produce a session key to encrypt the actual message.

We first describe some notation:

- G is a public group of prime order q generated by g.
- $V(h)$ takes a group element h and generates a random bit string from h. The function V is often called a key derivation function.
- H is a hash function producing an l-bit output.
- $y = g^x$ will be the public key corresponding to the private key x.

3.1.1. *Zheng–Seberry Scheme 1:* To encrypt a message m one computes

(1) $k \in_R \{1, \ldots, q-1\}$.
(2) $z = V(y^k)$.
(3) $t = H(m)$.
(4) $c_1 = g^k$.
(5) $c_2 = z \oplus (m \| t)$.
(6) Output (c_1, c_2).

When we decrypt a ciphertext we perform the following steps

(1) $z' = V({c_1}^x)$.
(2) $w = z' \oplus c_2$.
(3) t' is the last l bits of w.
(4) m' is the first $\#w - l$ bits of w.
(5) If $H(m') = t'$ then output m'.
(6) Output $\perp$.

The idea here is that we have added an extra piece of information to the ElGamal ciphertext, namely the encryption of the hash of the plaintext. Since it is meant to be hard to invert the hash function it should be hard to write down a valid ciphertext without knowing the corresponding plaintext. The addition of the hash adds the required redundancy which is then tested by the decryption function.

3.1.2. *Zheng–Seberry Scheme 2:* The second system uses a universal one-way hash function, which is essentially a parametrized set of hash functions H_i, where $i \leq \ell$. One can think of this in some ways as a keyed hash function or as a MAC.

More formally a universal one-way hash function is a keyed hash function H_k such that if the adversary is given x and then a hidden key k is chosen at random it should be hard for the adversary to be able to compute a y such that

$$H_k(y) = H_k(x).$$

To encrypt a message m in the second of Zheng and Seberry's schemes we compute

(1) $k \in_R \{1, \ldots, q-1\}$.
(2) Let z denote the $\#m$ leftmost bits of $V(y^k)$.
(3) Let s denote the ℓ rightmost bits of $V(y^k)$.
(4) $c_1 = g^k$.
(5) $c_2 = H_s(m)$.
(6) $c_3 = z \oplus m$.
(7) Output (c_1, c_2, c_3).

We leave it to the reader to write down the associated decryption function. The above system is similar to the first but the hash of the message is no longer encrypted, it is now sent in the clear.

But now one has an added difficultly since we do not know which hash function, or key, has been used to generate the hash value. The above system is very close to the system called DHIES which is currently considered the best practical algorithm based on ElGamal encryption.

3.1.3. *Zheng–Seberry Scheme 3:* In the third and final scheme produced by Zheng and Seberry one uses a DSA-like signature to 'sign' the message which is encrypted. The scheme works like a combination of an ElGamal-like encryption followed by a DSA signature, however the public key for the DSA signature is ephemeral and becomes part of the ciphertext. Again we leave it for the reader to write down the decryption algorithm.

(1) $k, t \in_R \{1, \ldots, q-1\}$.
(2) $r = y^{k+t}$.
(3) $z = V(r)$.
(4) $c_1 = g^k$.
(5) $c_2 = g^t$.
(6) $c_3 = (H(m) + xr)/k \pmod{q}$.
(7) $c_4 = z \oplus m$.
(8) Output (c_1, c_2, c_3, c_4).

Zheng and Seberry proved their schemes secure under a very strong conjectural assumption, namely that the space of ciphertexts was 'sole samplable'. This is an assumption akin to assuming that the encryption algorithm is plaintext aware. However, the first of the above schemes can be shown to be trivially insecure as follows. Suppose in the **find** stage our adversary outputs two messages m_1 and m_2. Then a hidden bit b is chosen and the adversary is given the encryption of m_b. This is equal to

$$c = (c_1, c_2) = \left(g^k, z \oplus (m_b \| H(m_b))\right).$$

The adversary in its **guess** stage can now perform the following operations. First it generates a new message m_3, different from m_1 and m_2, but of the same length. Then the adversary asks the decryption oracle to decrypt the ciphertext

$$(c_1, c_2 \oplus (m_1 \| H(m_1)) \oplus (m_3 \| H(m_3))).$$

When $b = 1$ the above ciphertext will be a valid encryption of m_3 and so the decryption oracle will return m_3. However, when $b = 0$ the above ciphertext is highly unlikely to be a valid encryption of anything, let alone m_3. This gives us a polynomial-time test, for an adaptive adversary, to detect the value of the hidden bit b.

3.2. RSA-OAEP. Recall that the raw RSA function does not provide a semantically secure encryption scheme, even against passive adversaries. To make a system which is secure we need either to add redundancy to the plaintext before encryption or to add some other form of redundancy to the ciphertext. In addition the padding used needs to be random so as to make a non-deterministic encryption algorithm. This is done in RSA by using a padding scheme, and over the years a number of padding systems have been proposed. However, some of the older ones are now considered weak.

By far the most successful padding scheme in use today was invented by Bellare and Rogaway and is called OAEP or Optimized Asymmetric Encryption Padding. OAEP is a padding scheme which can be used with any function which is a one-way trapdoor permutation, in particular the RSA function. When used with RSA it is often denoted RSA-OAEP.

Originally it was thought the RSA-OAEP was a plaintext-aware encryption algorithm, but this claim has since been shown to be wrong. However, one can show in the random oracle model that RSA-OAEP is semantically secure against adaptive chosen ciphertext attacks.

We first give the description of OAEP in general. Let f be any k-bit to k-bit trapdoor one-way permutation, e.g. for $k = 1024$ one could let f be the RSA function $c = m^e$. Let k_0 and k_1 denote numbers such that a work effort of 2^{k_0} or 2^{k_1} is impossible (e.g. $k_0, k_1 > 128$). Put $n = k - k_0 - k_1$ and let

$$G : \{0,1\}^{k_0} \longrightarrow \{0,1\}^{n+k_1}$$
$$H : \{0,1\}^{n+k_1} \longrightarrow \{0,1\}^{k_0}$$

be hash functions. Let m be a message of n bits in length. We then encrypt using the function

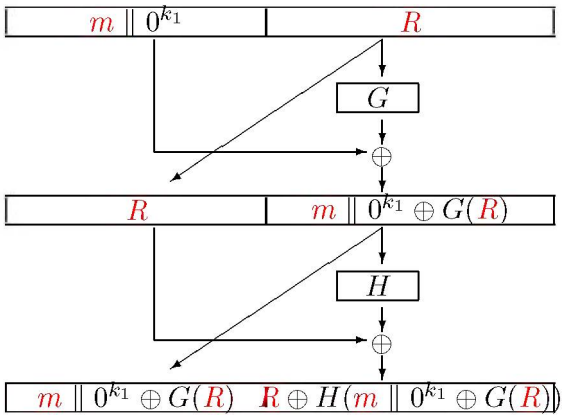
$$E(m) = f\left(\{m \parallel 0^{k_1} \oplus G(R)\} \| \{R \oplus H(m0^{k_1} \oplus G(R))\}\right).$$

where

- $m \parallel 0^{k_1}$ means m followed by k_1 zero bits,
- R is a random bit string of length k_0,
- $\parallel$ denotes concatenation.

One can view OAEP as a two-stage Feistel network as Fig. 1 demonstrates.

FIGURE 1. OAEP as a Feistel network



To decrypt a message given $E(m)$ and the trapdoor to f we can compute

$$A = \{T \| \{R \oplus H(T)\}\} = \left\{\{m \parallel 0^{k_1} \oplus G(R)\} \| \{R \oplus H(m \parallel 0^{k_1} \oplus G(R))\}\right\}.$$

So we know

$$T = m \parallel 0^{k_1} \oplus G(R).$$

Hence we can compute $H(T)$ and recover R from $R \oplus H(T)$. But then given R we can compute $G(R)$ and recover m. Note we need to check whether $T \oplus G(R)$ ends in k_1 zeros, if it does not then we should reject this ciphertext as invalid.

The main result about RSA-OAEP is:

THEOREM 20.5. *In the random oracle model, if we model G and H by random oracles then RSA-OAEP is semantically secure against adaptive chosen ciphertext attacks if the RSA assumption holds.*

PROOF. We sketch the proof and leave the details for the interested reader to look up. We first rewrite the RSA function f as

$$f : \begin{cases} \{0,1\}^{n+k_1} \times \{0,1\}^{k_0} \longrightarrow (\mathbb{Z}/N\mathbb{Z})^* \\ (s,t) \longmapsto (s\|t)^e \pmod N. \end{cases}$$

Note it is impossible to mathematically write the RSA function like this, but let us assume that we can. We then define RSA-OAEP as

$$s = (m\|0^{k_1}) \oplus G(r),\ t = r \oplus H(s).$$

The RSA assumption can be proved to be equivalent to the partial one-wayness of the function f, in the sense that the problem of recovering s from $f(s,t)$ is as hard as recovering (s,t) from $f(s,t)$. So for the rest of our sketch we try to turn an adversary A for breaking RSA-OAEP into an algorithm B_A which solves the partial one-wayness of the RSA function. In particular B_A is given $c^* = f(s^*,t^*)$, for some fixed RSA modulus N, and is asked to compute s^*.

Algorithm B_A now calls the **find** stage of A to produce two messages m_0 and m_1. A bit b is then chosen by B_A and B_A now assumes that c^* is the encryption of m_b. The ciphertext c^* is now given to the **guess** stage of A, and A tries to guess the bit b. Whilst algorithm A is running the algorithm B_A must answer the oracle queries for the hash function G, the hash function H and the decryption oracle. To maintain consistency, B_A keeps two lists, an H-List and a G-List of all prior calls to the oracles for H and G respectively.

The oracle queries are answered by B_A as follows:

- Query $G(\gamma)$:
  For any query δ in the H-List one checks whether
  $$c^* = f(\delta, \gamma \oplus H(\delta)).$$
  - If this holds then we have inverted f as required, we can still continue with the simulation of G and set
    $$G(\gamma) = \delta \oplus (m_b\|0^{k_1}).$$
  - If this equality does not hold for any value of δ then we choose $G(\gamma)$ uniformly at random from the codomain of G.
- Query $H(\delta)$:
  A random value is chosen from the codomain of H and set to this value. We also check whether for any γ in the G-List we have
  $$c^* = f(\delta, \gamma \oplus H(\delta)),$$
  if so we have managed to partially invert the function f as required.
- Query decryption of c :
  We look in the G-List and the H-List for a pair γ, δ such that if we set
  $$\sigma = \delta,\ \tau = \gamma \oplus H(\delta) \text{ and } \mu = G(\gamma) \oplus \delta,$$
  then $c = f(\sigma,\tau)$ and the k_1 least significant bits of μ are equal to zero. If this is the case then we return the plaintext consisting of the n most significant bits of μ, otherwise we return $\perp$.

Notice that if a ciphertext which was generated in the correct way (by calling G, H and the encryption algorithm) is then passed to the above decryption oracle, we will obtain the original plaintext back.

We have to show that the above decryption oracle is able to 'fool' the adversary A enough of the time. In other words when the oracle is passed a ciphertext, which had not been generated by a prior call to the necessary G and H, we need to show that it produces a value which is consistent with the running of the adversary A.

Finally we need to show that if the adversary A has a non-negligible chance of breaking the semantic security of RSA-OAEP then one has a non-negligible probability that B_A can partially invert f.

These last two facts are proved by careful analysis of the probabilities associated with a number of events. Recall that B_A assumes that $c^* = f(s^*, t^*)$ is an encryption of m_b. Hence, there should exist an r^* which satisfies

$$r^* = H(s^*) \oplus t^*,$$
$$G(r^*) = s^* \oplus (m_b \| 0^{k_1}).$$

One first shows that the probability of the decryption simulator failing is negligible. Then one shows that the probability that s^* is actually asked of the H oracle is non-negligible, as long as the adversary A has a non-negligible probability of finding the bit b. But as soon as s^* is asked of H then we spot this and can therefore break the partial one-wayness of f.

The actual technical probability arguments are rather involved and we refer the reader to the paper of Fujisaki, Okamoto, Pointcheval and Stern where the full proof is given. □

3.3. Turning CPA Schemes into CCA2 Schemes. Suppose we have a public key encryption scheme which is semantically secure against chosen plaintext attacks, such as ElGamal encryption. Such a scheme by definition needs to be non-deterministic hence we write the encryption function as

$$E(m, r),$$

where m is the message to be encrypted and r is the random input and we denote the decryption function by $D(c)$. Hence, for ElGamal encryption we have

$$E(m, r) = (g^r, m \cdot h^r).$$

Fujisaki and Okamoto showed how to turn such a scheme into an encryption scheme which is semantically secure against adaptive adversaries. Their result only applies in the random oracle model and works by showing that the resulting scheme is plaintext aware. We do not go into the details of the proof at all, but simply give the transformation which is both simple and elegant.

We take the encryption function above and alter it by setting

$$E'(m, r) = E(m \| r, H(m \| r))$$

where H is a hash function. The decryption algorithm is also altered in that we first compute

$$m' = D(c)$$

and then we check that

$$c = E(m', H(m')).$$

If this last equation holds we recover m from $m' = m \| r$, if the equation does not hold then we return $\perp$.

For ElGamal encryption we therefore obtain the encryption algorithm

$$(g^{H(m\|r)}, (m \| r) \cdot h^{H(m\|r)}),$$

which is only marginally less efficient than raw ElGamal encryption.

Chapter Summary

- The main technique in provable security is to show how the existence of an adversary against the cryptographic scheme under consideration can be used to solve some supposedly hard problem. Since one believes the problem to be hard, one then concludes that such an adversary cannot exist.
- The random oracle model is a computational model used as a proof technique in provable security. A proof in the random oracle model does not mean the system is secure in the real world, it only provides evidence that it may be secure.
- In the random oracle model one can use the forking lemma to show that certain discrete logarithm signature schemes are secure. To obtain proofs in the case of active adversaries one uses the random oracle to simulate the signing queries of the adversary.
- One can also show in the random oracle model that the two main RSA based signature schemes used in 'real life' are also secure, namely RSA-FDH and RSA-PSS.
- Proving encryption algorithms to be secure is slightly more tricky, early attempts of Zheng and Seberry made use of non-standard assumptions.
- In the random oracle model one can prove that the standard RSA encryption method, namely RSA-OAEP, is secure.

Further Reading

Provable security is a rapidly expanding field, and the number of papers grows with each passing year. A good description of the forking lemma and its applications is given in the article of Pointcheval and Stern. The random oracle model and a number of applications including RSA-FDH and RSA-PSS are given in the papers of Bellare and Rogaway. The full proof of the security of RSA-OAEP is given in the paper of Fujisaki and others.

M. Bellare and P. Rogaway. *Random oracles are practical: a paradigm for designing efficient protocols.* In Proc. 1st Annual Conf. on Comp. and Comms. Security, ACM, 62–73, 1993.

M. Bellare and P. Rogaway. *The exact security of digital signatures – How to sign with RSA and Rabin.* In Advances in Cryptology – EuroCrypt '96. Springer-Verlag LNCS 1070, 399–416, 1996.

E. Fujisaki, T. Okamoto, D. Pointcheval and J. Stern. *RSA-OAEP is secure under the RSA assumption.* In Advances in Cryptology – CRYPTO 2001, Springer-Verlag LNCS 2139, 260–274, 2001.

D. Pointcheval and J. Stern. *Security arguments for digital signatures and blind signatures.* J. Cryptology, **13**, 361–396, 2000.

CHAPTER 21

Hybrid Encryption

Chapter Goals

- To introduce the security notions for symmetric encryption schemes.
- To introduce and formalise the notion of hybrid encryption, via KEMs and DEMs.
- To present an efficient DEM, built out of a block cipher and a MAC.
- To present two efficient KEMs, namely RSA-KEM and DHIES.

1. Introduction

Almost always public key schemes are used to only transmit a short per message secret, such as a session key. This is because public key schemes are too inefficient to use to encrypt vast amounts of data. The actual data is then encrypted using a symmetric cipher. Such an approach is called a hybrid encryption scheme.

In this chapter we first define the security notions for symmetric ciphers which are analogous to those presented in Chapter 18. This allows us to then go on and formalise a way of designing a hybrid cipher, using the KEM/DEM approach. A KEM is a Key Encapsulation Mechansism, which is the public key component of a hybrid cipher, whilst a DEM is the Data Encapsulation Mechansism, which is the symmetric component. We show how a secure DEM can be formed from a weakly secure symmetric cipher and a MAC.

We then argue, without using random oracles, that a suitably secure DEM and a suitably secure KEM can be combined to produce a secure hybrid cipher. This means we only need to consider the symmetric and public key parts seperately, thus simplifying our design considerably.

Finally, we show how a KEM can be constructed in the random oracle model using either the RSA or the DLP primitive. The resulting KEMs are very simple to construct and very natural, and thus we see the simplification we obtain by considering hybrid encryption.

2. Security of Symmetric Ciphers

In this section we extend the security model of semantic security under adaptive chosen ciphertext attack for public key schemes to the symmetric setting, we then look at what the security definition should be for message authentication codes.

2.1. Security Models for Symmetric Ciphers. Semantic/polynomial security of symmetric ciphers is defined similarly to that of public key schemes. Let $E_k(m)$ denote the encryption algorithm and $D_k(c)$ denote the decryption algorithm.

The adversary runs in two stages:

- **Find**: In the find stage the adversary produces two plaintext messages m_0 and m_1, of equal length.
- **Guess**: The adversary is now given the encryption $c_b = E_k(m_b)$ of one of the plaintexts m_b for some secret hidden bit b under some randomly chosen, and secret, symmetric key

k. The goal of the adversary is to guess the value of the bit b with probability greater than one half.

Note, that this does not mean, as it does in the public key setting, that the encryption scheme must be probabilistic since the adversary is unable to evaluate the encryption algorithm without knowing the secret key k. Also, note that the key, under which the challeneg message is encrypted, is chosen after the adversary has chosen his two messages.

This type of security is often called *one-time* security, since the adversary only sees one valid ciphertext encrypted under the target key. This needs to be compared to the public key setting whereby an adversary can generate as many valid ciphertexts of its choosing by simply encrypting a message using the public key.

The advantage of an adversary A in the above game is defined to be

$$\mathrm{Adv}_A = \left| \Pr(A(\mathbf{guess}, c_b, m_0, m_1) = b) - \frac{1}{2} \right|.$$

A scheme is said to be secure if for all adversaries, all polynomials p and sufficiently large t we have

$$\mathrm{Adv}_A \leq \frac{1}{p(t)},$$

where t is the key size of the symmetric cipher. We say that Adv_A grows as a negligible function of t.

Notice, that in the above game that the adversary only sees one encryption of a message under the secret key k. Schemes which are secure in this sense are often called one-time encryption schemes. Recall that in Chapter 8 we commented that sometimes one used CBC mode with a fixed IV, this is precisely the situation we have here. Hence, for one-time encryption schemes one can use deterministic encryption.

To see a distinct difference between the public key and the private key setting, notice that in the public key setting the adversary could obtain encryptions of messages of her choice. This resulted in public key schemes needing to be probabilistic in nature. In the symmetric setting the adversary is not able to obtain encryptions of messages of her choice, bar the encryption of m_b. Thus we can use deterministic symmetric encryption in this setting.

The above definition defines the notion of semantic security (or indistinguishability of encryptions) against passive attacks. It is clear that the one-time pad will meet the above security definition. To define the notion for adaptive adversaries, i.e. CCA attacks, we give the adversary a decryption box in the second phase of the game which will decrypt, under the key k, arbitrary ciphertexts c of her choosing. Except the adversary is not allowed to query the decryption box with the target ciphertext c_b.

There are situations where one used a symmetric cipher in places where an adversary is able to obtain encryptions of messages of their choice. Such a situation is said to be a chosen plaintex attack, i.e. CPA attack. Note, that in the symmetric setting a CPA attack is an active attack whilst in the public key setting a CPA attack is a passive attack. In such situations the symmetric cipher needs to be probabilistic in nature. It is in these situations where one could use CBC Mode with a randomized IV.

It is assumed that a good block cipher used in CBC Mode produces a symmetric cipher which is semantically secure under passive attacks (fixed or random IV) or chosen plaintext attacks (random IV). However, they do not produce ciphers which are secure under chosen ciphertext attacks as the following example illustrates. Suppose the adversary selects two messages m_1 and m_2 of length twice the block length of the underlying cipher, i.e. we write

$$\begin{aligned} m_1 &= m_{1,1} \| m_{1,2}, \\ m_2 &= m_{2,1} \| m_{2,2}, \end{aligned}$$

where $m_{i,j}$ is a single block. We assume that $m_{1,1} \neq m_{2,1}$. The challenger then encrypts, using CBC Mode, one of these messages to produce the three block ciphertext

$$c_b = c_1 \| c_2 \| c_3,$$

where c_1 is the IV, and c_2 and c_3 result from encrypting the relavant blocks under CBC Mode.

The adversary then forms the ciphertext

$$c = c_1 \| c_2 \| c_3'$$

where c_3' is a random block. Since this ciphertext is different from the challenge ciphertext the adversary is allowed to ask her decryption box to decrypt it. When decrypted the resulting message will be of the following form

$$m_{b,1} \| m_2',$$

where m_2' is a random message block. Since the first block of the decryption will be equal to the first block of one of the original challenge messages, the adversary can determine which message was encrypted. Note the above chosen ciphertext attack also applies when the CBC Mode encryption is used with a random value of IV. Hence, the attack shows a distinct different between CPA and CCA attacks in the symmetric setting.

The reason why this attack works is that CBC Mode on its own contains no integrity checking, i.e. the adversary can alter blocks and will still obtain a valid ciphertext. A similar, but simpler, attack applies in the case of the one-time pad, which is secure against passive attacks but not against chosen ciphertext attacks. Our earlier public key encryption schemes which are secure against CCA attacks all had some notion of decryption failure, hence it appears that we need to extend this idea to the symmetric setting.

So whilst a standard block cipher is semantically secure against passive attacks, we need to use a different term to signal a symmetric cipher which is secure against active attacks. The term used to describe such symmetric ciphers is that of Data Encapsulation Mechanism, or DEM for short. We shall see that it is easy to construct such a DEM by taking a standard block cipher and appending a MAC to it. Before, presenting this we first need to define what we mean by a MAC being secure.

2.2. Security Models for MACs. Recall that a MAC is a function which takes as input a secret key k and a message m and outputs a code $c = MAC_k(m)$. Intuitively, a MAC should be secure if no-one can produce a valid MAC on a message without knowing the secret key k.

The weakest form of security for a MAC is one which for which an adversary cannot win the following game.

- **Stage 1**: The adversary first outputs a message m.
- **Stage 2**: The challenger chooses a secret MAC key k and returns $c = MAC_k(m)$ to the adversary. The adversaries goal is then to output a pair new MAC pair m', c' with $m' \neq m$ for which the MAC verifies.

Let Adv_A denote the advantage of the adversary in solving the above game. We require, for a MAC to be secure, that this advantage grows negligibly as a function of the bit length of k.

This definition can clearly be extended to allow the adversary to adaptively select other messages on which it wishes to obtain a MAC, and then it needs to output a MAC on a message which it has not seen. However, we shall only require the weak definition of security given above.

2.3. Constructing DEMs. Recall a DEM is a symmetric cipher which is semantically secure against adaptive chosen ciphertext adversaries. We can now turn to a standard construction of a DEM from a symmetric cipher E_k which is sematically secure against passive attacks and a MAC function MAC_k which is secure is the above sense. There are other ways of constructing secure DEMs, but the following is the simplest given constructions we have already met.

We build the DEM as follows: The key space of the DEM is the product of the key space of the symmetric cipher and the key space of the MAC. i.e. The DEM key consists of a pair of keys (k_1, k_2), one for the symmetric cipher and one for the MAC.

To encrypt a message m, one first forms the encryption $c_1 = E_{k_1}(m)$ under the symmetric cipher. Then one takes the MAC of the resulting ciphertext $c_2 = MAC_{k_2}(c_1)$ under the MAC. The output of the DEM is the pair (c_1, c_2).

To decrypt such a pair (c_1, c_2) one first checks whether the MAC verifies correctly, by checking whether $c_2 \stackrel{?}{=} MAC_{k_2}(c_1)$. If this fails then the decryptor outputs Invalid Ciphertext, otherwise the message is decrypted in the usual way via $D_{k_1}(c_1)$.

Notice, how the MAC is taken on the ciphertext and not the message and that there is the notion of invalid ciphertexts, which is similar to the cases we have looked at in the public key setting. However, we need to show that such a construction meets our security definition.

THEOREM 21.1. *The above construction of a DEM is sematically secure against active adversaries, asssuming the underlying symmetric cipher is semantically secure against passive adversaries and the MAC is secure.*

PROOF. As usual we only sketch the proof, the interested reader should consult the paper by Cramer and Shoup mentioned in the further reading of this chapter for more details.

We prove the theorem by assuming that there exists an adversary A against the DEM construction, and then showing that if this is successful either we can break the underlying symmetric cipher, or the MAC function.

We run A as a passive adversary against the underlying symmetric cipher, by responding to all its decryption queries as Invalid Ciphertext. Two possibilties occur, either A breaks the underlying symmetric cipher, or algorithm A realises it is interacting with an invalid decryption oracle. In the first case A has broken the symmetric cipher under a passive attack, in the second case we have managed to construct an adversary which breaks the underlying MAC, i.e. algorithm A has created a new MAC which verifies. □

3. Hybrid Ciphers

Since when encrypting large amounts of data we do not want to use a public key scheme it is common to encrypt the data with a symmetric cipher and then encrypt the key used for the symmetric cipher with a public key scheme. Thus we use the public key scheme as a key transport mechanism, like in Chapter 9. It turns out that if we carefully define what properties we want from such a scheme it is easier to define such mechanisms than full blown public key encryption schemes.

We define a Key Encapsulation Mechanism, or KEM, as a mechanism which from the encryptors side takes a public key y and outputs a symmetric key k and an encapsulation of that key c for use by the holder of the corresponding private key. The holder of the private key x can then take the encapsulation c and their private key, and then recover the symmetric key k. This is exactly the property one would require of a public key, key transport mechanism.

The idea of a KEM-DEM system is that one takes a KEM and a DEM, such that the symmetric keys output by the KEM match the key space of the DEM, and then use the two together to form a hybrid cipher. In more detail, suppose one wished to encrypt a message m to a user with public/private key pair (y, x). One would perform the following steps:

- $(k, c_1) = KEM(y)$.
- $c_2 = DEM_k(m)$.
- Send c_1, c_2 to the recipient.

The recipient would, upon recieving the pair c_1, c_2, perform the following steps, where $\perp$ denote the Invalid Ciphertext symbol,

- $k = KEM^{-1}(c_1, x)$.

- If $k = \perp$ return $\perp$.
- $m = DEM_k^{-1}(c_2)$.
- Return m

We would like the above hybrid cipher to meet our security definition for public key encryption schemes, namely semantic security against adaptive chosen ciphertext attacks. We shall see that the use of a DEM in the above hybrid cipher is crucial in meeting this strong definition. But before explaining why the above hybrid cipher is secure we need to define what it means for a KEM to be secure.

The security definition for KEMs is based on the security definition of indistinguishability of encryptions for public key encryption algorithms. However, we now require that the key output by a KEM should be indistinguishable from a random key. Thus the security game is defined via the following game.

- The challenger generates a random key k_0 from the space of symmetric keys output by the KEM.
- The challenger calls the KEM to produce a valid key k_1 and its encapsulation c under the public key y.
- The challenger picks a bit b and sends to the adversary the values k_b, c.
- The goal of the adversary is to decide whether $b = 0$ or 1.

The advantage of the adversary is defined to be

$$\mathrm{Adv}_A = \left| \Pr(A(k_b, c, y) = b) - \frac{1}{2} \right|.$$

If t is the security parameter, i.e. the size of the public key y, then the scheme is said to be secure if

$$\mathrm{Adv}_A \leq \frac{1}{p(t)},$$

for all adversaries A and all polynomials p and sufficiently large t.

The above only defines the security in the passive case, to define security under adaptive chosen ciphertext attacks one needs to give the adversary access to a decapsulation function. This decapsulation function will return the key (or the invalid encapsulation symbol) for any encapsulation of the adversaries choosing, bar the target encapsulation c. This decapsulation function can be called both before and after the adversary is given the challenge encapsulation c, in which case the security game above becomes a two-stage game as in the case of public key encryption schemes.

We can now show that if we use a suitably secure KEM and a suitably secure DEM, then the resulting hybrid cipher is secure.

THEOREM 21.2. *If one uses a KEM and a DEM which are secure against active adversaries in creating the hybrid cipher, then the hybrid cipher is a public key encryption scheme which had indistinguishable encryptions in the presence of adaptive chosen ciphertext attacks.*

We do not give the proof, but direct the reader to the paper of Cramer and Shoup for the details. The proof is relatively simple, however it uses a technique called game-hopping which we have not introduced in this book.

4. Constructing KEMs

One of the benefits we said about using KEMs was we said that they were easier to design. In this section we first look at RSA-KEM, whose construction and proof should be compared to that of RSA-OAEP. Then we turn to DHIES which should be compared to either the Zheng–Seberry schemes mentioned earlier, or the discrete logarithm scheme based on the Fujisaki–Okamoto transform in Chapter 20.

4.1. RSA-KEM. Let N denote an RSA modulus, i.e. a product of two primes p and q of roughly the same size. Let e denote an RSA public exponent and d an RSA private exponent. We let $f_N(x)$ denote the RSA function, i.e. the function that maps an integer x modulo N to the number $x^e \pmod N$. Recall, that the RSA is the problem of given an integer y modulo N to recover the value of x such that $f_N(x) = y$.

We define RSA-KEM by taking a cryptographic hash function H which takes integers modulo N and maps them to symmetric keys, of the size required by the user of the KEM (i.e. the key size of the DEM. Encapsulation then works as follows:

- Generate $x \in \{1, \ldots, N-1\}$.
- Compute $c = f_N(x)$.
- Compute $k = H(x)$.
- Output (k, c).

Since the person with the private key can invert the function f_N, decapsulation is easily performed via

- Compute $x = f_N^{-1}(c)$.
- Compute $k = H(x)$.
- Output k.

Note, how there is no notion of invalid ciphertexts and how simple this is in comparison to RSA-OAEP. We only need show that this simple construction meets our definition of a secure KEM

THEOREM 21.3. *In the random oracle model RSA-KEM is a secure KEM in the context of active adversaries, assuming the RSA problem is hard.*

PROOF. Given an adversary A we wish to construct an algorithm B_A which solves the RSA problem. We model H via a random oracle, thus B_A keeps a list of triples (x, c, h) of queries to H, which is initialy set to be empty.

The adversary is allowed to make queries of H. If this query, on x, has been made before then B_A uses its list to respond as required. If there is a value on the list of the form $(\perp, c, h)$ with $f_N(x) = c$ then B_A replaces this value with (x, c, h) and responds with h. Otherwise B_A generates a new random value of h, adds the triple $(x, f(x), h)$ to the list and responds with h.

The adversary can also make decapsulation queries on an encapsulation c. To answer these B_A checks whether $f_N(x) = c$ for a value $(x, *, h)$ on the list, if it is B_A replaces this entry with (x, c, h) and we respond with h. If there is a value $(*, c, h)$ on the list it responds with h. Otherwise, it generates h at random, places the triple $(\perp, c, h)$ on the list and responds with h.

At some point A will request its challenge encapsulation. At this point B_A generates a symmetric key k at random and takes as the challenge encapsulation the value y of the RSA function which B_A is trying to find the preimage of. It then passes k and y to A.

Since A is running in the random oracle model, the only way that A can have any success in the game is by querying H on the preimage of y. Thus if A is successful then the preimage of y will exist on the list of triples kept by algorithm B_A. □

4.2. The DHIES Encryption Scheme. The DHIES encryption scheme is very similar to the type of immunization techniques originally proposed by Zheng and Seberry. However, one can prove that DHIES is secure against adaptive chosen ciphertext attack, assuming the three components are themselves secure, the three components being

- A finite cyclic abelian group in which the interactive Hash Diffie–Hellman (HDH) assumption holds.
- A symmetric key based encryption function which is semantically secure against adaptive chosen plaintext attacks.

- A message authentication code, or keyed hash function, for which an active adversary cannot find the MAC of an 'unasked message'. One should think of this last concept as a MAC version of existential forgery.

The DHIES scheme of Bellare and Rogaway integrates all three of the above components giving rise to the schemes name, Diffie–Hellman Integrated Encryption Scheme. Originally the scheme was called DHAES, for Diffie–Hellman Augmented Encryption Scheme, but this caused confusion with the Advanced Encryption Standard.

Key generation for DHIES is just like that of ElGamal encryption. The domain parameters are a cyclic finite abelian group G of prime order q, a generator g, a symmetric encryption function $\{E_k, D_k\}$, a MAC function which we shall denote by MAC_k and a hash function H.

To generate a public/private key pair we generate a random $x \in \mathbb{Z}/q\mathbb{Z}$ and compute the public key

$$f = g^x.$$

To encrypt a message m we generate a random per message ephemeral key k and compute

$$v = f^k \text{ and } u = g^k.$$

Then we pass this into the hash function to obtain two keys, one for the symmetric encryption algorithm and one for the MAC function,

$$(k_1, k_2) = H(u\|v).$$

We then compute

$$c = E_{k_1}(m),$$
$$t = MAC_{k_2}(c).$$

The ciphertext is transmitted as the triple

$$u\|c\|t.$$

Notice that this scheme allows arbitrary long messages to be encrypted efficiently using a public key scheme. The u acts as a key transport mechanism, the encryption is performed using a symmetric encryption function, the protection against adaptive adversaries for the whole scheme is provided by the addition of the MAC t.

On receiving the ciphertext $u\|c\|t$ the legitimate private key holder can compute

$$v = u^x.$$

Then the two keys k_1 and k_2 can be computed via

$$(k_1, k_2) = H(u\|v).$$

The ciphertext is then rejected if the MAC does not verify, i.e. we should have

$$t = MAC_{k_2}(c).$$

Finally, the plaintext is recovered from

$$m = D_{k_1}(c).$$

We note that the scheme follows our KEM-DEM paradigm in fact one can derive the DHIES scheme by defining a KEM and then using our earlier hybrid technique along with out construction of a DEM from a block cipher and a MAC. We shall follow this technique as we can then show that the full DHIES scheme is secure by applying Theorems 21.1 and 21.2.

The DHIES KEM is then defined as follows: We generate a random per message ephemeral key k and compute

$$v = f^k \text{ and } u = g^k.$$

The secret key output by the KEM is the value of

$$H(u\|v),$$

whilst the encapsulation of this key is the value u. To decapsulate the KEM one takes u and using the private key one computes

$$v = u^x.$$

Then the secret key can be recovered using the hash function just as the sender did.

However, to prove this KEM secure we need to introduce a new problem called the Gap-Diffie–Hellman problem. This problem assumes that the Diffie–Hellman problem is hard even assuming that the adversary has an oracle to solve the decision Diffie–Hellman problem. In other words, we are given g^a and g^b and a function F which on input of (g^x, g^y, g^z) will say whether $z = x \cdot y$. We then wish to output g^{ab}. It is believed that this problem is as hard as the standard Diffie–Hellman problem. Indeed there are some groups in which the decisional Diffie–Hellman problem is easy and the computational Diffie–Hellman problem is hard.

We can now prove that the DHIES KEM is secure:

THEOREM 21.4. *In the random oracle model and assuming the Gap-Diffie–Hellman problem is hard, there exists no adversary which breaks the DHIES KEM.*

PROOF. Again we give a sketch. Assume A is an adversary against the KEM, we wish to construct an algorithm B_A which solves the Gap-Diffie–Hellman problem. Let $f = g^a$ and $c^* = g^b$ be in the inputs to algorithm B_A. We then let f denote the public key of the scheme, and we let c^* denote the target encapsulation and let k denote a random key chosen by B_A. Algorithm B_A then calls the KEM adversary with the inputs (f, c^*, k).

This adversary will make a number of hash function and decapsulation queries. The algorithm B_A responds to these queries as follows. To simulate the hash function algorithm B_A maintains a list of values (h, c, x). Suppose that the hash function is called on the value x'. There are three cases which can occur.

- If x' occurs as the third component in an item in this list then the corresponding value h' is returned to the adversary A.
- If x' is such that there is an item on the list of the form $(h', c', \perp)$ such that (c', f, x') is a valid Diffie–Hellman tuple (which can be determined from the DDH oracle given to B_A), then $(h', c', \perp)$ is replaced by (h', c', x') and h' is returned to the adversary A.
- Otherwise h' is generated at random from the range of the hash function, the triple $(h', \perp, x')$ is placed on the list and h' is returned to A.

The decapsulation oracle queries are also handled via the use of the list used to simulate the hash function and the use of the DDH oracle. Suppose that the adversary A calls its decapsulation oracle on the input c'. There are three cases which can occur.

- If c' occurs as the second component in an item in the list list then the corresponding value h' is returned to the adversary A.
- If c' is such that there is an item on the list of the form $(h', \perp, x')$ such that (c', f, x') is a valid Diffie–Hellman tuple (which can be determined from the DDH oracle given to B_A), then $(h', \perp, x')$ is replaced by (h', c', x') and h' is returned to the adversary A.
- Otherwise h' is generated at random from the range of the hash function, the triple $(h', c', \perp)$ is placed on the list and h' is returned to A.

It is easy to see that in the random oracle model the adversary A cannot tell the difference between this simulation of the decapsulation and hash function queries and a genuine simulation.

Also because we are in the random oracle model the only way that the adversary can tell whether k is a validly encapsulated by c^* is if it calls the hash function on the Diffie–Hellman value g^{ab}. Thus, once A terminates algorithm B_A finds the correct value of g^{ab} since it must be one of the third components of an item on its list. Indeed, the exact value can be found via the use of the DDH oracle on all items in the list. □

Chapter Summary

- We presented the KEM-DEM paradigm for designing public key encryption schemes.
- We described the security model for symmetric ciphers and showed how to construct ciphers which are secure against active attacks.
- We gave the RSA-KEM and showed why it is secure, assuming the RSA problem is hard.
- We presented DHIES-KEM and showed why it is secure, assuming the Gap-Diffie–Hellman problem is hard.

Further Reading

The paper by Cramer and Shoup presents the subject matter of this chapter in great detail, it is recommended for further study of this area. The DHIES scheme was first presented in the paper by Abdalla et. al. A good paper to look at for various KEM constructions is that by Dent.

M. Abdalla, M. Bellare and P. Rogaway. *DHAES: An encryption scheme based on the Diffie–Hellman problem.* Submission to IEEE P1363a standard.

R. Cramer and V. Shoup. *Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack.* SIAM Journal of Computing **33**, 167–226, 2003.

A. Dent. *A designer's guide to KEMs.* In Cryptography and Coding – 2003, Springer-Verlag LNCS 2898, 133–151, 2003.

CHAPTER 22

Provable Security: Without Random Oracles

Chapter Goals

- To cover some of the more modern schemes which can be proved secure without recourse to the random oracle model.
- To examine the strong RSA assumption and the interactive Hash Diffie–Hellman assumption.
- To explain the GHR and Cramer–Shoup signature algorithms.
- To explain the Cramer–Shoup encryption algorithm.

1. Introduction

In the previous chapter we looked at signature and encryption schemes which can be proved secure in the so-called 'random oracle model'. The random oracle model does not model the real world of computation. A proof in the random oracle model only provides evidence that a scheme may be secure in the real world, it does not guarantee security in the real world. We can interpret a proof in the random oracle model as saying that if an adversary against the real-world scheme exists then that adversary must make use of the specific hash function employed.

In this chapter we sketch recent work on how researchers have tried to construct signature and encryption algorithms which do not depend on the random oracle model. We shall only consider schemes which are practical, and we shall only sketch the proof ideas. Readers interested in the details of proofs or in other schemes should consult the extensive literature in this area.

What we shall see is that whilst quite natural encryption algorithms can be proved secure without the need for random oracles, the situation is quite different for signature algorithms. This is the opposite case to what we saw when we used random oracles to model hash functions. In the previous chapter it was the signature algorithms which looked more natural compared with the encryption algorithms. This should not be surprising, signature algorithms make extensive use of hash functions for their security. Hence, we should expect that they impose stricter restraints on such hash functions, which may not be actually true in the real world.

However, the removal of the use of random oracles comes at a price. We need to make stronger intractability assumptions than we have otherwise made. In the next section we outline a new intractability assumptions. This new assumption is related to an assumption we have met before, namely that the RSA problem is hard, but the new problem is much less studied than the ones we have met before. It is also also a potentially easier problem than ones we have met before, hence the assumption that the problem is hard is a much stronger assumption than before.

2. The Strong RSA Assumption

We have already met and studied the RSA assumption, which is the assumption that the following problem is hard:

DEFINITION 22.1 (RSA Problem). *Given an RSA modulus* $N = p \cdot q$, *an exponent* e *with* $\gcd(e, \phi(N)) = 1$ *and a random* $c \in (\mathbb{Z}/N\mathbb{Z})^*$ *find* $m \in (\mathbb{Z}/N\mathbb{Z})^*$ *such that*

$$m^e = c.$$

The *strong RSA assumption* is the assumption that the following problem is hard:

DEFINITION 22.2 (Flexible RSA Problem). *Given an RSA modulus* $N = p \cdot q$ *and a random* $c \in (\mathbb{Z}/N\mathbb{Z})^*$ *find* $e > 1$ *and* $m \in (\mathbb{Z}/N\mathbb{Z})^*$ *such that*

$$m^e = c.$$

Clearly if we can solve the RSA problem then we can solve the flexible RSA problem. This means that the strong RSA assumption is a stronger intractability assumption than the standard RSA assumption, in that it is conceivable that in the future we may be able to solve the flexible RSA problem but not the RSA problem proper. However, at present we conjecture that both problems should be equally as hard.

3. Signature Schemes

We have already remarked that signature schemes which are provably secure, without the random oracle model are hard to come by. They also appear somewhat contrived compared with the schemes such as RSA-PSS, DSA or Schnorr which are used in real life. The first such provably secure signature scheme in the standard model was by Goldwasser, Micali and Rivest. This was however not very practical as it relied on messages being associated with leaves of a binary tree, and each node in the tree needed to be authenticated with respect to its parent. This made the resulting scheme far too slow.

In this section we consider two modern provably secure signature schemes which are 'practical'. However, we shall see with both of them that they still come with some problems which do not occur in more standard signature schemes.

3.1. Gennaro–Halevi–Rabin Signature Scheme. In 1999 Gennaro, Halevi and Rabin came up with a provably secure signature scheme, called the GHR signature scheme, which does not require the random oracle model for its security. The security of this scheme is based on the strong RSA assumption.

As the key generation step one takes an RSA modulus

$$N = p \cdot q$$

where p and q are chosen to be safe primes, in that both $(p-1)/2$ and $(q-1)/2$ should also be prime. This restriction on p and q implies that finding an odd integer which is not co-prime to

$$\phi(N) = (p-1)(q-1)$$

is as hard as factoring. In addition to N the public key also consists of a random element $s \in (\mathbb{Z}/N\mathbb{Z})^*$.

To sign a message m the signer, who knows the factors of N, can compute a value σ such that

$$\sigma^{H(m)} = s \pmod{N}.$$

The above equation serves also as the verification equation.

One can see how the scheme is related to the strong RSA assumption immediately. However, the scheme requires a very special type of hash function. To see why, suppose an active adversary wanted a signature on a message m_1 and he could find another message m_2 such that

$$H(m_2) = a \cdot H(m_1).$$

Now suppose the active attacker asked the signer for a signature on m_2. The signer outputs σ_2 such that

$$\sigma_2{}^{H(m_2)} = s \pmod{N}.$$

The attacker can now forge a signature on the message m_1 by computing

$$\sigma_1 = \sigma_2^a \pmod{N},$$

since

$$\begin{aligned}\sigma_1{}^{H(m_1)} &= (\sigma_2^a)^{H(m_1)} \pmod{N}\\ &= \sigma_2{}^{aH(m_1)} \pmod{N}\\ &= \sigma_2{}^{H(m_2)} \pmod{N}\\ &= s.\end{aligned}$$

The authors of the above signature scheme propose that the easiest way to avoid this type of attack would be for the hash function to be such that it always returned a random prime number. One would still require that finding collisions in the hash function was hard. Whilst it is possible to design hash functions which only output prime numbers they are not well studied and are not 'natural' hash functions.

3.2. The Cramer–Shoup Signature Scheme. The Cramer–Shoup signature scheme is still based on the strong RSA assumption and is provably secure, without the need for the random oracle model. Like the previous GHR signature scheme the signer needs to generate prime numbers, but these do not have to be the output of the hash function. Hence, the Cramer–Shoup scheme can make use of a standard hash function, such as SHA-1. In our discussion below we shall assume H is a 'standard' hash function which outputs bit strings of 160 bits, which we interpret as 160-bit integers as usual.

Again to generate the public key, we create an RSA modulus N which is the produce of two safe primes p and q. We also choose two random elements

$$h, x \in Q_N,$$

where as usual Q_N is the set of quadratic residues modulo N. We also create a random 160-bit prime e'. The public key consists of

$$(N, h, x, e')$$

whilst the private key is the factors p and q.

To sign a message the signer generates another 160-bit prime number e and another random element $y' \in Q_N$. The signer then computes, since they know the factors of N, the solution y to the equation

$$y = \left(x h^{H(x')}\right)^{1/e} \pmod{N},$$

where x' satisfies

$$x' = y'^{e'} h^{-H(m)}.$$

The output of the signer is

$$(e, y, y').$$

To verify a message the verifier first checks that e' is an odd number satisfying

$$e \neq e'.$$

Then the verifier computes

$$x' = y'^{e'} h^{-H(m)}$$

and then checks that

$$x = y^e h^{-H(x')}.$$

On the assumption that H is a collision resistant hash function and the strong RSA assumption holds, one can prove that the above scheme is secure against active adversaries. We sketch the most important part of the proof, but the full details are left to the interested reader to look up.

Assume the adversary makes t queries to a signing oracle. We want to use the adversary A to create an algorithm B_A to break the strong RSA assumption for the modulus N. Before setting up the public key for input to algorithm A, the algorithm B_A first decides on what prime values e_i it will output in the signature queries. Then, having knowledge of the e_i, the algorithm B_A concocts values for the h and x in the public key, so that it always knows the e_ith root of h and x.

So when given a signing query for a message m_i, algorithm B_A can then compute a valid signature, without knowing the factorization of N, by generating $y'_i \in Q_N$ at random and then computing

$$x'_i = y'^{e'}_i h^{-H(m_i)} \pmod{N}$$

and then

$$y_i = x^{1/e_i} (h^{1/e_i})^{H(x'_i)} \pmod{N},$$

the signature being given by

$$(m_i, y_i, y'_i).$$

The above basic signing simulation is modified in the full proof, depending on what type of forgery algorithm A is producing. But the basic idea is that B_A creates a public key to enable it to respond to every signing query in a valid way.

4. Encryption Algorithms

Unlike the case of signature schemes, for encryption algorithms one can produce provably secure systems which are practical and close to those used in 'real life', without assuming the random oracle model.

4.1. The Cramer–Shoup Encryption Scheme. The Cramer–Shoup encryption scheme requires as domain parameters a finite abelian group G of prime order q. In addition we require a universal one-way family of hash functions. Recall, this is a family $\{H_i\}$ of hash functions for which it is hard for an adversary to choose an input x, then to draw a random hash function H_i, and then to find a different input y so that

$$H_i(x) = H_i(y).$$

A public key in the Cramer–Shoup scheme is chosen as follows. First the following random elements are selected

$$g_1, g_2 \in G,$$
$$x_1, x_2, y_1, y_2, z \in \mathbb{Z}/q\mathbb{Z}.$$

The user then computes the following elements

$$c = {g_1}^{x_1} {g_2}^{x_2},$$
$$d = {g_1}^{y_1} {g_2}^{y_2},$$
$$h = {g_1}^{z}.$$

The user finally chooses a hash function H from the universal one-way family of hash functions and outputs the public key

$$(g_1, g_2, c, d, h, H),$$

whilst keeping the private key secret

$$(x_1, x_2, y_1, y_2, z).$$

The encryption algorithm proceeds as follows, which is very similar to ElGamal encryption. The message m is considered as an element of G. The sender then chooses a random ephemeral key $r \in \mathbb{Z}/q\mathbb{Z}$ and computes

$$\begin{aligned} u_1 &= {g_1}^r, \\ u_2 &= {g_2}^r, \\ e &= m \cdot h^r, \\ \alpha &= H(u_1 \| u_2 \| e), \\ v &= c^r d^{r\alpha}. \end{aligned}$$

The ciphertext is then the quadruple

$$(u_1, u_2, e, v).$$

On receiving this ciphertext the owner of the private key can recover the message as follows: First they compute $\alpha = H(u_1 \| u_2 \| e)$ and test whether

$${u_1}^{x_1 + y_1 \alpha} {u_2}^{x_2 + y_2 \alpha} = v.$$

If this equation does not hold then the ciphertext should be rejected. If this equation holds then the receiver can decrypt the ciphertext by computing

$$m = \frac{e}{{u_1}^z}.$$

To show that the scheme is provably secure, under the assumption that the DDH problem is hard and that H is chosen from a universal one-way family of hash functions, we assume we have an adversary A against the scheme and show how to use A in another algorithm B_A which tries to solve the DDH problem.

One way to phrase the DDH problem is as follows: Given $(g_1, g_2, u_1, u_2) \in G$ determine whether either this quadruple is a random quadruple or we have $u_1 = {g_1}^r$ and $u_2 = {g_2}^r$ for some value of $r \in \mathbb{Z}/q\mathbb{Z}$. So algorithm B_A will take as input a quadruple $(g_1, g_2, u_1, u_2) \in G$ and try to determine whether this is a random quadruple or a quadruple related to the Diffie–Hellman problem.

Algorithm B_A first needs to choose a public key, which it does in a non-standard way, by first selecting the random elements

$$x_1, x_2, y_1, y_2, z_1, z_2 \in \mathbb{Z}/q\mathbb{Z}.$$

Algorithm B_A then computes the following elements

$$\begin{aligned} c &= {g_1}^{x_1} {g_2}^{x_2}, \\ d &= {g_1}^{y_1} {g_2}^{y_2}, \\ h &= {g_1}^{z_1} {g_2}^{z_2}. \end{aligned}$$

Finally B_A chooses a hash function H from the universal one-way family of hash functions and outputs the public key

$$(g_1, g_2, c, d, h, H).$$

Notice that the part of the public key corresponding to h has been chosen differently than in the real scheme, but that algorithm A will not be able to detect this change.

Algorithm B_A now runs the **find** stage of algorithm A, responding to decryption queries of (u_1', u_2', e', v') by computing

$$m = \frac{e'}{{u_1'}^{z_1} {u_2'}^{z_2}},$$

after performing the standard check on validity of the ciphertext. The output of the **find** stage will be two plaintexts m_0, m_1.

After running the **find** stage, algorithm B_A chooses a bit b at random and computes the target ciphertext as

$$\begin{aligned} e &= m_b \cdot (u_1^{z_1} u_2^{z_2}), \\ \alpha &= H(u_1 \| u_2 \| e), \\ v &= u_1^{x_1 + y_1 \alpha} u_2^{x_2 + y_2 \alpha}. \end{aligned}$$

The target ciphertext is then the quadruple

$$(u_1, u_2, e, v).$$

Notice that when the input to B_A is a legitimate DDH quadruple then the target ciphertext will be a valid encryption, but when the input to B_A is not a legitimate DDH quadruple then the target ciphertext is highly likely to be an invalid ciphertext.

This target ciphertext is then passed to the **guess** stage of the adversary A. If this adversary outputs the correct value of b then we suspect that the input to B_A is a valid DDH quadruple, whilst if the output is wrong then we suspect that the input to B_A is not valid. This produces a statistical test to detect whether the input to B_A was valid or not. By repeating this test a number of times we can produce as accurate a statistical test as we want.

Note, that the above is only a sketch. We need to show that the view of the adversary A in the above game is no different from that in a real attack on the system, otherwise A would know something was not correct. For example we need to show that the responses B_A makes to the decryption queries of A cannot be distinguished from a true decryption oracle. For further details one should consult the full proof in the paper mentioned in the Further Reading section.

Notice that, whilst very similar to ElGamal encryption, the Cramer–Shoup encryption scheme is much less efficient. Hence, whilst provably secure it is not used much in practice due to the performance disadvantages.

Chapter Summary

- Signature schemes which are secure without the random oracle model are less natural than those which are secure in the random oracle model. This is probably because signatures make crucial use of hash functions which are easy to model by random oracles.
- The strong RSA assumption is a natural weakening of the standard RSA assumption.
- The Cramer–Shoup encryption scheme is provably secure, without the random oracle model, assuming the DDH problem is hard. It is around three times slower than the usual ElGamal encryption algorithm.

Further Reading

The schemes mentioned in this chapter can be found in the following papers.

M. Abdalla, M. Bellare and P. Rogaway. *DHAES: An encryption scheme based on the Diffie–Hellman problem.* Submission to IEEE P1363a standard.

R. Cramer and V. Shoup. *A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack.* In Advances in Cryptology – CRYPTO '98, Springer-Verlag LNCS 1462, 13–25, 1998.

R. Cramer and V. Shoup. *Signature schemes based on the strong RSA assumption.* ACM Transactions on Information and Systems Security, **3**, 161–185, 2000.

R. Gennaro, S. Halevi and T. Rabin. *Secure hash-and-sign signatures without the random oracle.* In Advances in Cryptology – EuroCrypt '99, Springer-Verlag LNCS 1592, 123–139, 1999.

Part 5

Advanced Protocols

Encryption, Hash Functions, MACs and Signatures are only the most basic of cryptographic constructions and protocols. We usually think of them as being carried out between a sender and a receiver who have the same security goals. For example in encryption, both the sender and the receiver probably wish to keep the message secret from an adversary. In other words the adversary is assumed to be someone else.

In this section we shall detail a number of more advanced protocols. These are mainly protocols between two or more people, in which the security goals of the different parties could be conflicting, or different. For example in an electronic election voters want their votes to be secret, yet all parties want to know that all votes have been counted, and all parties want to ensure against a bad voter casting too many votes. Hence, the adversaries are also the parties in the protocol, they are not necessarily external entities.

First we focus on secret sharing schemes, which allow a party to share a secret amongst a number of partners. This has important applications in splitting of secrets into parts which can then be used in distributed protocols. Then we turn to commitments schemes and oblivious transfer. These are two types of basic protocols between two parties, in which the parties are assumed to be mutually untrustworthy, i.e. the adversary is the person who you are performing the protocol with. We then turn to the concept of zero-knowledge proofs. In this chapter we also examine a simple electronic voting scheme. Finally we look at the subject of secure multi-party computation, which provides an interesting application of many of our proceeding algorithms.

CHAPTER 23

Secret Sharing Schemes

Chapter Goals

- To introduce the notion of secret sharing schemes.
- To give some simple examples for general access structures.
- To present Shamir's scheme, including how to recover the secret in the presence of active adversaries.
- To show the link between Shamir's secret sharing and Reed–Solomon codes.

1. Introduction

Suppose you have a secret s which you wish to share amongst n parties $\mathcal{P}$. You would like certain subsets of the n parties to recover the secret but not others. The classic scenario might be that s is nuclear launch code and you have four people, the president, the vice-president, the secretary of state and a general in a missile silo. You do not want the general to be able to launch the missile without the president agreeing, but to maintain deterrence you would like, in the case the president has been eliminated, that the vice-president, the secretary of state and the general can agree to launch the missile. If we label the four parties as P, V, S and G, for president, vice-president, secretary of state and general, then we would like the following sets of people to be able to launch the missile

$$\{P, G\} \text{ and } \{V, S, G\},$$

but no smaller sets. It is this problem which secret sharing is designed to deal with, however the applications are more widespread than might at first appear.

To each party we distribute some information called the *share*. For a party A we will let s_A denote the secret share which they hold. In the example above there are four such shares s_P, s_V, s_S and s_G. Then if the required parties come together we would like an algorithm which combines their relevant shares into the secret s. It is this problem which this chapter addresses.

2. Access Structures

Before introducing schemes to perform secret sharing we first however, need to introduce the notion of an access structure. Any subset of parties who can recover the secret will be called a *qualifying set*, whilst the set of all qualifying sets will be called an *access structure*. So in the example above we have that the two sets

$$\{P, G\} \text{ and } \{V, S, G\}$$

are qualifying sets. However, clearly any set containing such a qualifying set is also a qualifying set. Thus

$$\{P, G, V\},\ \{P, G, S\} \text{ and } \{P, V, G, S\}$$

are also qualifying sets. Hence, there are five sets in the access structure. For any set in the access structure, if we have the set of shares for that set we would like to be able to reconstruct the secret.

DEFINITION 23.1. *Consider a set $\mathcal{P}$. A monotone structure on $\mathcal{P}$ is a collection Γ of subsets of $\mathcal{P}$ such that*

- $\mathcal{P} \in \Gamma$
- *If $A \in \Gamma$ and B is a set such that $A \subset B \subset \mathcal{P}$ then $B \in \Gamma$.*

Thus in the above example the access structure is monotone. This is a property which will hold for all access structures of all secret sharing schemes. For a monotone structure we note that the sets in Γ come in chains, $A \subset B \subset C \subset \mathcal{P}$. We shall call the sets which form the start of a chain, the *minimal qualifying sets*. The set of all such minimal qualifying sets for an access structure Γ we shall denote by $m(\Gamma)$. We can now give a very informal definition of what we mean by a secret sharing scheme:

DEFINITION 23.2. *A secret sharing scheme for a monotone access structure Γ over a set of parties $\mathcal{P}$ with respect to a space of secrets S, is a pair of algorithms called* `Share` *and* `Recombine` *with the following properties:*

- `Share`(s, Γ) *takes a secret s and a monotone access structure and determines a value s_A for every $A \in \mathcal{P}$. The value s_A is called A's share of the secret.*
- `Recombine`(H) *takes a set H of shares for some set $\mathcal{O}$ of $\mathcal{P}$, i.e.*

$$H = \{s_O : O \in \mathcal{O}\}.$$

If $\mathcal{O} \in \Gamma$ then this should return the secret s, otherwise it should return nothing.

A secret sharing scheme is considered to be secure if no adversary can learn anything about the underlying secret without having access to the shares of a qualifying set. Actually such schemes are said to be information theoretically secure, but since most secret sharing schemes in the literature are information theoretically secure we shall just call such schemes *secure*.

In this chapter we will consider two running examples of monotone access structures, so as to illustrate the schemes. Both will be on sets of four elements: The first is from the example above where we have $\mathcal{P} = \{P, V, S, G\}$ and

$$\Gamma = \{\{P, G\}, \{V, S, G\}, \{P, G, V\}, \{P, G, S\}, \{P, V, G, S\}\}\,.$$

The set of minimal qualify sets are given by

$$\{\{P, G\}, \{V, S, G\}\}\,.$$

The second example we shall define over the set of parties $\mathcal{P} = \{A, B, C, D\}$, with access structure

$$\begin{aligned}\Gamma = \{\,&\{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\},\\ &\{A, B, C\}, \{A, B, D\}, \{B, C, D\}, \{A, B, C, D\}\,\}\,.\end{aligned}$$

The set of minimal qualify sets are given by

$$\{\{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\}\}\,.$$

This last access structure is interesting because it represents a common structure of a threshold access structure. Notice that we require that any two out of the four parties should be able to recover the secret. We call such a scheme a 2-out-of-4 threshold access structure.

One way of looking at such access structures is via a boolean formulae. Consider the set $m(\Gamma)$ of minimal qualifying sets and define the formulae:

$$\bigvee_{\mathcal{O} \in m(\Gamma)} \left(\bigwedge_{O \in \mathcal{O}} O \right).$$

For example in our first example above the formulae becomes

$$(P \wedge G) \vee (V \wedge S \wedge G).$$

Reading this formulae out, with $\wedge$ being "and", and $\vee$ being "or", we see that one can reconstruct the secret if we have access to the secret shares of

(P and G) or (V and S and G).

Notice how the formula is in disjunctive normal form (DNF).

3. General Secret Sharing

We now turn to two methods for constructing secret sharing schemes for arbitrary monotone access structures. They are highly inefficient for all but the simplest access structures but they do show that one can cope with an arbitrary access structure. We assume that the space of secrets S is essentially the set of bit strings of length n-bits. In both examples we let $s \in S$ denote the secret which we are trying to share.

3.1. Ito-Nishizeki-Saito Secret Sharing. Our first secret sharing scheme makes use of the DNF boolean formulae we presented above. In some sense every "or" gets converted into a concatenation operation and every "and" gets converted into a $\oplus$ operation, note this can at first sight seem slightly counter-intuitive.

The sharing algorithm works as follows. For every minimal qualifying set $\mathcal{O} \in m(\Gamma)$, we generate shares $s_i \in S$, for $1 \leq i \leq l$, at random, where $l = |\mathcal{O}|$ such that $s_1 \oplus \cdots \oplus s_l = s$. Then a party A is given a share s_i if it occurs as position i in the set $\mathcal{O}$.

3.1.1. *Example 1:* Recall we have the formulae

(P and G) or (V and S and G).

We generate shares five elements s_i from S such that

$$\begin{aligned} s &= s_1 \oplus s_2, \\ &= s_3 \oplus s_4 \oplus s_5. \end{aligned}$$

The four shares are then defined to be:

$$\begin{aligned} s_P &= s_1, \\ s_V &= s_3, \\ s_S &= s_4, \\ s_G &= s_2 \| s_5. \end{aligned}$$

You should check that, given this sharing, any qualifying set can recover the secret, and only the qualifying sets can recover the secret. Notice, that party G needs to hold two times more data than the size of the secret. Thus this scheme in this case is not efficient. Ideally we would like the parties to only hold the equivalent of n bits of information each so as to recover a secret of n bits.

3.1.2. *Example 2:* Now our formulae is given by.

(A and B) or (A and C) or (A and D) or (B and C) or (B and D) or (C and D).

We now generate shares for twelve elements s_i from S, one for each of the distinct terms in the formulae above, such that

$$\begin{aligned} s &= s_1 \oplus s_2, \\ &= s_3 \oplus s_4, \\ &= s_5 \oplus s_6, \\ &= s_7 \oplus s_8, \\ &= s_9 \oplus s_{10}, \\ &= s_{11} \oplus s_{12}. \end{aligned}$$

The four shares are then defined to be:

$$\begin{aligned} s_A &= s_1 \| s_3 \| s_5, \\ s_B &= s_2 \| s_7 \| s_9, \\ s_C &= s_4 \| s_8 \| s_{11}, \\ s_D &= s_6 \| s_{10} \| s_{12}. \end{aligned}$$

You should again check that, given this sharing, any qualifying set and only the qualifying sets can recover the secret. We see that in this case every share contains three times more information than the underlying secret.

3.2. Replicated Secret Sharing. The above is not the only scheme for general access structure. Here we present another one called the replicated secret sharing scheme. In this scheme we first create the sets of all maximal non-qualifying sets, these are the sets of all parties such that if you add a single new party to each set you will obtain a qualifying set. If we label these sets $A_1, \ldots, A_t$, we then form their set-theoretic complements, i.e. $B_i = \mathcal{P} \setminus A_i$. A set of secret shares s_i is then generated, one for each set B_i, so that

$$s = s_1 \oplus \cdots \oplus s_t.$$

Then a party is given the share s_i if it is contained in the set B_i.

3.2.1. *Example 1:* The sets of maximal non-qualifying sets for this example are

$$A_1 = \{P, V, S\}, \ A_2 = \{V, G\} \text{ and } A_3 = \{S, G\}$$

Forming their complements we obtain the sets

$$B_1 = \{G\}, \ B_2 = \{P, S\} \text{ and } B_3 = \{P, V\}.$$

We generate three shares s_1, s_2 and s_3 such that $s = s_1 \oplus s_2 \oplus s_3$ and then define the shares as

$$\begin{aligned} s_P &= s_2 \| s_3, \\ s_V &= s_3, \\ s_S &= s_2, \\ s_G &= s_1. \end{aligned}$$

Again we can check that only the qualifying sets can recover the secret.

3.2.2. *Example 2:* For the 2-out-of-4 threshold access structure we obtain the following maximal non-qualifying sets

$$A_1 = \{A\},\ A_2 = \{B\},\ A_3 = \{C\} \text{ and } A_4 = \{D\}.$$

On forming their complements we obtain

$$B_1 = \{B, C, D\},\ B_2 = \{A, C, D\},\ B_3 = \{A, B, D\} \text{ and } B_4 = \{A, B, C\}.$$

We form the four shares such that $s = s_1 \oplus s_2 \oplus s_3 \oplus s_4$ and

$$\begin{aligned} s_A &= s_2 \| s_3 \| s_4, \\ s_B &= s_1 \| s_3 \| s_4, \\ s_C &= s_1 \| s_2 \| s_4, \\ s_D &= s_1 \| s_2 \| s_3. \end{aligned}$$

Whilst the above two constructions provide a mechanism to construct a secret sharing scheme for any monotone access structure, they appear to be very inefficient. In particular for the threshold access structure they are particularly bad, especially as the number of parties increases. In the rest of this chapter we will examine a very efficient mechanism for threshold secret sharing due to Shamir, called Shamir secret sharing. This secret sharing scheme is itself based on the ideas behind certain error-correcting codes, called Reed–Solomon codes. So we will first have a little digression into coding theory.

4. Reed–Solomon Codes

An error-correcting code is a mechanism to transmit data from A to B such that any errors which occur during transmission, for example due to noise, can be corrected. They are found in many areas of electronics; they are the thing which makes your CD/DVD resistant to minor scratches, they make sure that RAM chips preserve your data correctly, they are used for communication between earth and satellites or deep space probes.

A simpler problem is one of error-detecting. Here one is only interested in whether the data has been altered or not. A particularly important distinction to make between the area of coding theory and cryptography is that in coding theory one can select simpler mechanisms to detect errors. This is because, in coding theory the assumption is that the errors are introduced by random noise, whereas in cryptography any errors are thought to be actively inserted by an adversary. Thus in coding theory, error detection mechanisms can be very simple, whereas in cryptography we have to resort to complex mechanism's such as MAC's and digital signatures.

Error correction on the other hand is not only interested with detecting errors, it also wants to correct those errors. Clearly one cannot correct all errors, but it would be nice to correct a certain number. A classic way of forming error-correcting codes is via Reed–Solomon codes. Usually such codes are presented, in coding theory, over a finite field of characteristic two. However, we are interested in the general case and so we will be using a code over $\mathbb{F}_q$, for a prime power q.

To define a Reed–Solomon code we also require two other integer parameters n and t. The value n defines the length of each code-word, whilst the number t is related to the number of errors we can correct. We also define a set $X \subset \mathbb{F}_q$ of size n. If the characteristic of $\mathbb{F}_q$ is larger than n then we can select $X = \{1, 2, \ldots, n\}$, although any set will do. For our application to Shamir secret sharing later on we will assume that $0 \notin X$.

Consider the set of polynomials of degree less than or equal to t over the field $\mathbb{F}_q$.

$$\mathcal{P} = \left\{ f_0 + f_1 X + \ldots + f_t X^t : f_i \in \mathbb{F}_q \right\}.$$

The set $\mathcal{P}$ represents the total number of code-words in our code, i.e. the amount of different data which we can transmit in any given block. The total number of code-words is then equal to q^{t+1}.

To create the actual code-word we evaluate the polynomial at all elements in X. Hence, the set of actual code-words is given by

$$\mathcal{C} = \{(f(x_1), \ldots, f(x_n)) : f \in \mathcal{P}, x_i \in X\}.$$

The length of a code word is then given by $n \cdot \log_2 q$. So we require $n \cdot \log_2 q$ bits to represent $(t+1) \cdot \log_2 q$ bits of information.

As an example consider the Reed–Solomon code with parameters $q = 101$, $n = 7$, $t = 2$ and $X = \{1, 2, 3, 4, 5, 6, 7\}$. Suppose our "data", which is represented by an element of $\mathcal{P}$, is given by the polynomial

$$f = 20 + 57X + 68X^2.$$

To transmit this data we compute $f(i) \pmod P$ for $i = 1, \ldots, 7$, to obtain the code-word

$$c = (44, 2, 96, 23, 86, 83, 14).$$

This code-word can now be transmitted or stored.

4.1. Data Recovery. At some point the code-word will need to be converted back into the data. In other words we have to recover the polynomial in $\mathcal{P}$ from the set of points at which it was evaluated, i.e. the vector of values in $\mathcal{C}$. We will first deal with the simple case and assume that no errors have occurred.

The receiver is given the data $c = (c_1, \ldots, c_n)$ but has no idea as to the underlying polynomial f. Thus from the receivers perspective he wishes to find the f_i such that

$$f = \sum_{j=0}^{t} f_j X^j.$$

It is well known, from high school, that a polynomial of degree at most t is determined completely by its values at $t+1$ points. So as long as $t < n$ we can recover f when no errors occur; the question is how?

First note that the receiver can generate n linear equations via

$$c_i = f(x_i) \text{ for } x_i \in X.$$

In other words he has the system of equations:

$$\begin{aligned} c_1 &= f_0 + f_1 x_1 + \cdots + f_t x_1^t, \\ &\vdots \qquad\qquad \vdots \\ c_n &= f_0 + f_1 x_1 + \cdots + f_t x_n^t. \end{aligned}$$

So by solving this system of equations over the field $\mathbb{F}_q$ we can recover the polynomial and hence the data.

Actually the polynomial f can be recovered without the need for solving the linear system, via the use of Lagrange interpolation. Suppose we first compute the polynomials

$$\delta_i(X) = \prod_{x_j \in X, j \neq i} \frac{X - x_j}{x_i - x_j}, \quad 1 \leq i \leq n.$$

Note that we have the following properties, for all i.

- $\delta_i(x_i) = 1$.
- $\delta_i(x_j) = 0$, if $i \neq j$.
- $\deg \delta_i(X) = n - 1$.

Lagrange interpolation takes the values c_i and computes

$$f(X) = \sum_{i=1}^{n} c_i \cdot \delta_i(X).$$

The three properties above on the polynomials $\delta_i(X)$ translate into the following facts above $f(X)$

- $f(x_i) = c_i$ for all i.
- $\deg f(X) \leq n - 1$.

Hence, Lagrange interpolation finds the unique polynomial which interpolates the n elements in the code-word.

4.2. Error Detection. Returning to our Reed–Solomon codes we see that by using Lagrange interpolation on the code-word we will recover a polynomial of degree t when their are no errors, but when there are errors in the received code-word we are unlikely to obtain a valid polynomial, i.e. an element of $\mathcal{P}$. Hence, we instantly have an error-detection algorithm.

Returning to our example parameters above. Suppose the following code-word was received

$$c = (44, 2, 25, 23, 86, 83, 14).$$

In other words it is equal to the sent code-word except in the third position where a 96 has been replaced by a 25. We compute once and for all the polynomials, modulo $q = 101$,

$$\begin{aligned}
\delta_1(X) &= 70X^6 + 29X^5 + 46X^4 + 4X^3 + 43X^2 + 4X + 7,\\
\delta_2(X) &= 85X^6 + 12X^5 + 23X^4 + 96X^3 + 59X^2 + 49X + 80,\\
\delta_3(X) &= 40X^6 + 10X^5 + 83X^4 + 23X^3 + 48X^2 + 64X + 35,\\
\delta_4(X) &= 14X^6 + 68X^5 + 33X^4 + 63X^3 + 78X^2 + 82X + 66,\\
\delta_5(X) &= 40X^6 + 90X^5 + 99X^4 + 67X^3 + 11X^2 + 76X + 21,\\
\delta_6(X) &= 85X^6 + 49X^5 + 91X^4 + 91X^3 + 9X^2 + 86X + 94,\\
\delta_7(X) &= 70X^6 + 45X^5 + 29X^4 + 60X^3 + 55X^2 + 43X + 1.
\end{aligned}$$

The receiver now tries to recover the sent polynomial given the data he has received. He obtains

$$f(X) = 44 \cdot \delta_1(X) + \cdots + 14 \cdot \delta_7(X) = 60 + 58X + 94X^2 + 84X^3 + 66X^4 + 98X^5 + 89X^6.$$

But this is a polynomial of degree six and not of degree $t = 2$. Hence, the receiver knows that there is at least one error in the code word that he has received. He just does not know which one is in error, nor what its actual value should be.

4.3. Error Correction. The intuition behind error correction is the following. Consider a polynomial of degree three over the reals evaluates in seven points, such as that in Figure 1. Clearly there is only one cubic curve which interpolates all of the points, since we have specified seven of them and we only need four such points to define a cubic curve. Now suppose one of these evaluations is given in error, for example the point at $x = 3$, as in Figure 2. We see that we still have six points on the cubic curve, and so there is a unique cubic curve passing through these six valid points. However, if we took another six points, i.e. five valid ones and the error one, then it is highly likely that the curve which goes through second six points would not be cubic.

In other words because we have far more valid points than we need to determine the cubic curve, we are able to recover it. This leads to a very inefficient method for polynomial reconstruction. We return to the general case of a polynomial of degree t evaluated at n points. Suppose we know that there are at most e errors in our code-word. Then we simply produce the list of all subsets S of the n points with $n - e$ members. We then try to recover a polynomial of degree t, if we are successful then we have a good probability that the subset S is the valid set, if we are unsuccessful then we know that S contains an element which is in error.

FIGURE 1. Cubic function evaluated at seven points

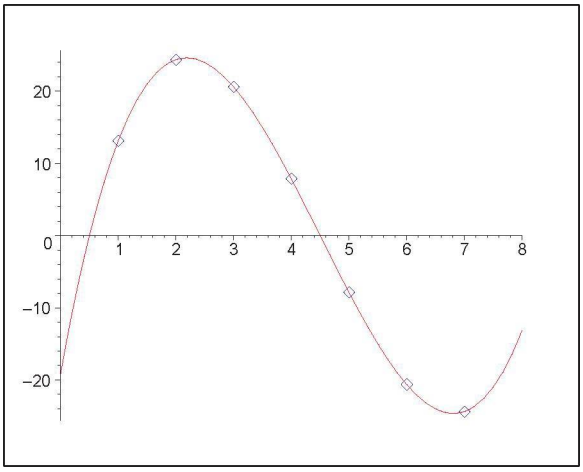
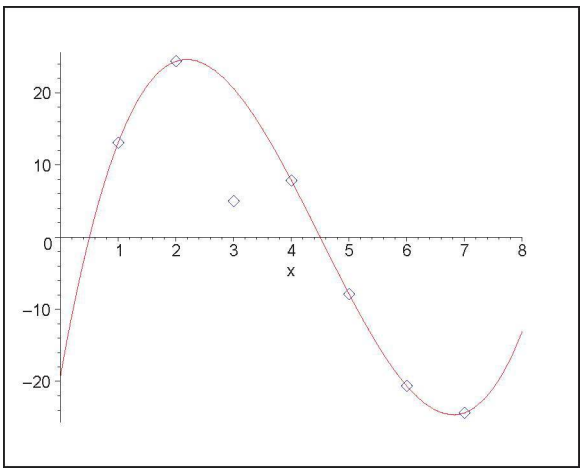


FIGURE 2. Cubic function going through six points and one error point



This is clearly a silly algorithm to error correct a Reed–Solomon code. The total number of subsets we may have to take is given by

$$^{n}C_{n-e} = \frac{n!}{e! \cdot (n-e)!}.$$

But we will still analyse this algorithm a bit more:

To be able to recover a polynomial of degree t we must have that $t < n - e$, i.e. we must have more valid elements than there are coefficients to determine. Suppose we not only have e errors but we also have s "erasures", i.e. values which we do not even receive. To recover a polynomial of degree t we will require

$$t < n - e - s.$$

But we could recover many such polynomials, for example if $t = n - e - s + 1$ then all such sets S will result in a polynomial of degree at most t. To obtain a *unique* polynomial from the above method we will need to make sure we do not have too many errors.

It can be shown that if we can obtain at least $t + 2e$ valid points then one can recover a unique polynomial of degree t which passes through $n-s-e$ points of the set of $n-s$ points. This gives the

important equation that an error correction for Reed–Solomon codes can be performed uniquely provided

$$n > t + 2e + s.$$

The only problem left is how to perform this error correction *efficiently*.

4.4. The Berlekamp–Welch Algorithm. We now present an efficient method to perform error correction for Reed–Solomon codes called the Berlekamp–Welch algorithm. The idea is to interpolate a polynomial in two variables through the points which we are given. Suppose we are given a code word with s missing values, and the number of errors is bounded by

$$e < t < \frac{n-s}{3}.$$

This means we are actually given $n - s$ supposed values of $y_i = f(x_i)$. We know the pairs (x_i, y_i) and we know that at most e of them are wrong, in that they do not come from evaluating the hidden polynomial $f(X)$. The goal of error correction is to try to recover this hidden polynomial.

We first construct the bivariate polynomial

$$Q(X, Y) = f_0(X) - f_1(X) \cdot Y,$$

where f_0 (resp. f_1) is a polynomial of degree at most $2 \cdot t$ (resp. t). We impose the condition that $f_1(0) = 1$. We treat the coefficients of the f_i as variables which we want to determine. Due to the bounds on the degrees of the two polynomials, and the extra condition of $f_1(0) = 1$, we see that the number of variables we have is

$$v = (2 \cdot t + 1) + (t + 1) - 1 = 3 \cdot t + 1.$$

We would like the bivariate polynomial $Q(X, Y)$ to interpolate our points (x_i, y_i). By substituting in the values of x_i and y_i we obtain a linear equation in terms of the unknown coefficients of the polynomials f_i. Since we have $n - s$ such points, the number of linear equations we obtain is $n - s$.

After determining f_0 and f_1 we then compute

$$f = \frac{f_0}{f_1}.$$

To see this consider the single polynomial in one variable

$$P(X) = Q(X, f(X))$$

where $f(X)$ is the polynomial we are trying to determine. We have $\deg P(X) \leq 2t$. The polynomial $P(X)$ clearly has at least $n - s - e$ zero's, i.e. the number of valid pairs. So the number of zeros is at least

$$n - s - e > n - e - t > 3t - t = 2t,$$

since $e < t < \frac{n-s}{3}$. Thus $P(X)$ has more zeros than it's degree, and it must hence be the zero polynomial. Hence,

$$f_0 - f_1 \cdot f = 0$$

and so $f = f_0/f_1$ since $f_1 \neq 0$.

Again consider our previous example, we have received the invalid code-word

$$c = (44, 2, 25, 23, 86, 83, 14).$$

We know that the underlying code is for polynomials of degree $t = 2$. Hence, since $2 = t < \frac{n-s}{3} = 7/3 = 2.3$ we should be able to correct a single error. Using the method above we want to determine the polynomial $Q(X, Y)$ of the form

$$Q(X, Y) = f_{0,0} + f_{1,0}X + f_{2,0}X^2 + f_{3,0}X^3 + f_{4,0}X^4 - \left(1 + f_{1,1}X + f_{2,1}X^2\right) Y$$

which passes through the seven given points. Hence we have six variables to determine and we are given seven equations. These equations form the linear system, modulo $q = 101$,

$$\begin{pmatrix} 1 & 1 & 1^2 & 1^3 & 1^4 & -44\cdot 1 & -44\cdot 1^2 \\ 1 & 2 & 2^2 & 2^3 & 2^4 & -2\cdot 2 & -2\cdot 2^2 \\ 1 & 3 & 3^2 & 3^3 & 3^4 & -25\cdot 3 & -25\cdot 3^2 \\ 1 & 4 & 4^2 & 4^3 & 4^4 & -23\cdot 4 & -23\cdot 4^2 \\ 1 & 5 & 5^2 & 5^3 & 5^4 & -86\cdot 5 & -86\cdot 5^2 \\ 1 & 6 & 6^2 & 6^3 & 6^4 & -83\cdot 6 & -83\cdot 6^2 \\ 1 & 7 & 7^2 & 7^3 & 7^4 & -14\cdot 7 & -14\cdot 7^2 \end{pmatrix} \cdot \begin{pmatrix} f_{0,0} \\ f_{1,0} \\ f_{2,0} \\ f_{3,0} \\ f_{4,0} \\ f_{1,1} \\ f_{2,1} \end{pmatrix} = \begin{pmatrix} 44 \\ 2 \\ 25 \\ 23 \\ 86 \\ 83 \\ 14 \end{pmatrix}.$$

So we are solving the system

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 57 & 57 \\ 1 & 2 & 4 & 8 & 16 & 97 & 93 \\ 1 & 3 & 9 & 27 & 81 & 26 & 78 \\ 1 & 4 & 16 & 64 & 54 & 9 & 36 \\ 1 & 5 & 25 & 24 & 19 & 75 & 72 \\ 1 & 6 & 36 & 14 & 84 & 7 & 42 \\ 1 & 7 & 49 & 40 & 78 & 3 & 21 \end{pmatrix} \cdot \begin{pmatrix} f_{0,0} \\ f_{1,0} \\ f_{2,0} \\ f_{3,0} \\ f_{4,0} \\ f_{1,1} \\ f_{2,1} \end{pmatrix} = \begin{pmatrix} 44 \\ 2 \\ 25 \\ 23 \\ 86 \\ 83 \\ 14 \end{pmatrix} \pmod{101}.$$

We obtain the solution

$$(f_{0,0}, f_{1,0}, f_{2,0}, f_{3,0}, f_{4,0}, f_{1,1}, f_{2,1}) = (20, 84, 49, 11, 0, 67, 0).$$

So we obtain the two polynomials

$$f_0(X) = 20 + 84X + 49X^2 + 11X^3 \text{ and } f_1(X) = 1 + 67X.$$

We find that

$$f(X) = \frac{f_0(X)}{f_1(X)} = 20 + 57X + 68X^2.$$

Which is precisely the polynomial we started with at the beginning of this section. Hence, we have corrected for the error in the transmitted code-word.

5. Shamir Secret Sharing

We now return to secret sharing schemes, and in particular the Shamir secret sharing scheme. We suppose we have n parties who wish to share a secret so that no t (or less) parties can recover the secret. Hence, this is going to be a $(t+1)$-out-of-n secret sharing scheme.

First we suppose there is a trusted dealer who wishes to share the secret s in $\mathbb{F}_q$. He first generates a secret polynomial $f(X)$ of degree t with $f(0) = s$. That is he generates random integers f_i in $\mathbb{F}_p$ for $i = 1, \ldots, t$ and sets

$$f(X) = s + f_1 X + \cdots + f_t X^t.$$

The trusted dealer then identifies each of the n players by an element in a set $X \subset \mathbb{F}_q \setminus \{0\}$, for example we could take $X = \{1, 2, \ldots, n\}$. Then if $i \in X$, party i is given the share $s_i = f(i)$.

Notice that the vector

$$(s_1, \ldots, s_n)$$

is a code-word for a Reed–Solomon code. Also note that if $t+1$ parties come together then they can recover the original polynomial via Lagrange interpolation and hence the secret s. Actually, secret reconstruction can be performed more efficiently by making use of the equation

$$s = f(0) = \sum_{i=1}^{n} s_i \delta_i(0).$$

Hence, if we define for a set $Y \subset X$ the vector r_Y by $r_Y = (r_{x_i,Y}, \ldots, r_{x_i,Y})_{x_i \in Y}$ to be the public "recombination" vector, where

$$r_{x_i,Y} = \prod_{x_j \in Y, x_j \neq x_i} \frac{-x_j}{x_i - x_j}.$$

Then, if we obtain a set of shares back from a subset $Y \subset X$ with $\#Y > t$, we can recover s via the simple summation.

$$s = \sum_{x_i \in Y} r_{x_i,Y} \cdot s_i.$$

Also note that if we receive from a set of parties Y some possible share values, then we can recover the original secret via the Berlekamp–Welch algorithm for decoding Reed-Solomon code in the presence of errors. Assuming the number of errors is bounded by e where

$$e < t < \frac{\#Y}{3}.$$

An interesting property of Shamir secret sharing is that we can do without the trusted dealer completely, in the case when $t = n - 1$, i.e. the case when we want all parties to come together to recover the secret. Suppose we have n parties and they wish to share a secret, but it does not matter what the secret actually is, only that it is shared in a n-out-of-n secret sharing scheme. To do this they all simply generate a random integer s_i as their share. These will define a polynomial of degree $n - 1$, and the resulting polynomial will be able to be recovered when all parties come together.

Shamir secret sharing is also an example of a scheme which can be made into a pseudorandom secret sharing scheme, or PRSS. This is a secret sharing scheme which allows the parties to generate a sharing of a random value with almost no interaction. For an n-out-of-n scheme this is simple since each party only needs to generate a random value, and then the common shared secret value is the sum of all the various shares. However, we wish to do this for a general $(t + 1)$-out-of-n scheme.

To create such a scheme for the Shamir secret sharing scheme requires some initial interaction to perform the setup, then the parties can generate as many shares of random values as they wish, with no further interaction. To define the Shamir pseudorandom secret sharing scheme we take our n parties, labelled by the set $X = \{1, 2, \ldots, n\}$, and threshold value t. Then for every subset $A \subset X$ of size $n - t$ we define the polynomial $f_A(X)$ of degree t by the conditions

$$f_A(0) = 1 \text{ and } f_A(i) = 0 \text{ for all } i \in X \setminus A.$$

In the initialisation phase for our PRSS we create a secret value $r_A \in S$, where S is some key space, for each subset A, this is securely distributed to each set A. The n parties also agree on a public pseudorandom function which is keyed by the secret values r_A,

$$\psi : \begin{cases} S \times S & \longrightarrow & \mathbb{F}_q \\ (r_A, x) & \longmapsto & \psi(r_A, x). \end{cases}$$

Now when the parties wish to generate a new secret sharing of a random value. By some means, either by interaction or by prearrangement (e.g. a counter value), they select a public random value $a \in S$. The underlying polynomial of degree t is given by

$$f(X) = \sum_{A \subset \{1,\ldots,n\}, |A| = n-t} \psi(r_A, a) \cdot f_A(X),$$

where the sum is over all subsets A of size $n - t$. This means that each party i receives the share

$$s_i = \sum_{i \in A \subset \{1,\ldots,n\}, |A| = n-t} \psi(r_A, a) \cdot f_A(i)$$

where the sum is over all subsets A of size $n-t$ which contain the element i. Finally the random value which is shared, via the Shamir secret sharing scheme, is given by

$$s = \sum_{A \subset \{1,\ldots,n\}, |A|=n-t} \psi(r_A, a) \cdot f_A(0) = \sum_{A \subset \{1,\ldots,n\}, |A|=n-t} \psi(r_A, a).$$

In Chapter 26 we shall require not only pseudorandom secret sharing, but also a variant called pseudorandom zero sharing, or PRZS, for the Shamir secret sharing scheme. In pseudorandom zero sharing we wish to generate random sharings of the value zero, with respect to a polynomial of degree $2 \cdot t$. For this we require exactly the set up as for the PRSS, but now we a different pseudorandom function,

$$\psi : \begin{cases} S \times S \times \{1, \ldots, t\} & \longrightarrow & \mathbb{F}_q \\ (r_A, x, j) & \longmapsto & \psi(r_A, x, j). \end{cases}$$

Then to create a degree $2 \cdot t$ Shamir secret sharing of zero, the parties pick a number a as before. The underlying polynomial is then given by

$$f(X) = \sum_{A \subset \{1,\ldots,n\}, |A|=n-t} \left(\sum_{j=1}^{t} \psi(r_A, a) \cdot X^j \cdot f_A(X) \right).$$

Clearly this is a polynomial which shares the zero value.

6. Application: Shared RSA Signature Generation

We shall now present a simple application of a secret sharing scheme, which has applications in the real world. Suppose a company is setting up a certificate authority to issue RSA signed certificates to its employees to enable them to access various corporate services. It considers the associated RSA private key to be highly sensitive, after all if the private key was compromised then the entire companies corporate infrastructure could also be compromised. Suppose the public key is (N, e) and the private key is d.

The company decide that to mitigate the risk they will divide the private key into three shares and place the three shares on three different continents. Thus, for example, there will be one server in Asia, one in America and one in Europe. As soon as the RSA key is generated the company generates three integers d_1, d_2 and d_3 such that

$$d = d_1 + d_2 + d_3 \pmod{\phi(N)}.$$

The company then removes all knowledge of d and places d_1 on a secure computer in Asia, d_2 on a secure compute in America and d_3 on a secure computer in Europe.

Now an employee wishes to obtain a digital certificate. This is essentially the RSA signature on a (possibly hashed) string m. The employee simply sends the string m to the three computers, who respond with

$$s_i = m^{d_i} \text{ for } i = 1, 2, 3.$$

The valid RSA signature is then obtained by multiplying the three shares together, i.e

$$s = s_1 \cdot s_2 \cdot s_3 = m^{d_1+d_2+d_3} = m^d.$$

This scheme appears to solve the problem of putting the master signature key in only one location. However, the employee now needs for the three servers to be online in order to obtain his certificate. It would be much nicer if only two had to be online, since then the company could cope with outages of servers.

The problem is that the above scheme essentially implements a 3-out-of-3 secret sharing scheme, whereas what we want is a 2-out-of-3. Clearly, we need to apply something along the lines of Shamir

secret sharing. However, the problem is that the number $\phi(N)$ needs to be kept secret, and the denominators in the Lagrange interpolation formulae may not be coprime to $\phi(N)$.

There have been many solutions proposed to the above problem of threshold RSA, however, the most elegant and simple is due to Shoup. Suppose we want a t-out-of-n sharing of the RSA secret key d, where we assume that e is chosen so that it is a prime and $e > n$. We adapt the Shamir scheme as follows: Firstly a polynomial of degree $t-1$ is chosen, by selecting f_i modulo $\phi(N)$ at random, to obtain

$$f(X) = d + f_1 X + \cdots + f_{t-1} X^{t-1}.$$

Then each server is given the share $d_i = f(i)$. The number of parties n is assumed to be fixed and we set $\Delta = n!$.

Now suppose a user wishes to obtain a signature on the message m, i.e it wants to compute $m^d \pmod N$. It sends m to each server, which then computes the signature fragment as

$$s_i = m^{2\Delta d_i} \pmod N.$$

These signature fragments are then sent back to the user.

Suppose now that the user obtains fragments back from a subset $Y = \{i_1, \ldots, i_t\} \subset \{1, \ldots, n\}$, of size greater than or equal to t. Consider the "recombination" vector defined by

$$r_{i_j,Y} = \prod_{i_k \in Y, i_j \neq i_k} \frac{-i_k}{i_j - i_k}.$$

We really want to be able to compute this modulo $\phi(N)$ but that is impossible. However we note that the denominator in the above divides Δ and so we have that $\Delta \cdot r_{i_j,Y} \in \mathbb{Z}$. Hence, the user can compute

$$s' = \prod_{i_j \in Y} s_{i_j}^{2 \cdot \Delta \cdot r_{i_j,Y}} \pmod N.$$

We find that this is equal to

$$\begin{aligned} s' &= \left(m^{4 \cdot \Delta^2}\right)^{\sum_{i_j \in Y} r_{i_j,Y} \cdot d_{i_j}} \\ &= m^{4 \cdot \Delta^2 \cdot d} \pmod N. \end{aligned}$$

With the last equality working due to Lagrange interpolation modulo $\phi(N)$. From this partial signature we need to recover the real signature. To do this we use the fact that we have assumed that $e > n$ and e is a prime. These latter two facts mean that e is coprime to $4 \cdot \Delta^2$, and so via the extended Euclidean algorithm we can compute integers u and v such that

$$u \cdot e + v \cdot 4 \cdot \Delta^2 = 1.$$

From which the signature is computed as

$$s = m^u \cdot s'^v \pmod N.$$

That s is the valid RSA signature for this public/private key pair can be verified since

$$\begin{aligned} s^e &= \left(m^u \cdot s'^v\right)^e, \\ &= m^{e \cdot u} \cdot m^{4 \cdot e \cdot v \cdot \Delta^2 \cdot d}, \\ &= m^{u \cdot e + 4 \cdot v \cdot \Delta^2}, \\ &= m. \end{aligned}$$

Chapter Summary

- We have defined the general concept of secret sharing schemes and shown how these can be constructed, albeit inefficiently, for any access structure.
- We have introduced Reed–Solomon error correcting codes and presented the Berlekamp–Welch decoding algorithm.
- We presented Shamir's secret sharing scheme which produces a highly efficient, and secure, secret sharing scheme in the case of threshold access structures.
- We extended the Shamir scheme to give both pseudorandom secret sharing and pseudorandom zero sharing.
- Finally we showed how one can adapt the Shamir scheme to enable the creation of a threshold RSA signature scheme.

Further Reading

Shamir's secret sharing scheme is presented in his short ACM paper from 1983. Shoup's threshold RSA scheme is presented in his EuroCrypt 2000 paper, this paper also explains the occurance of the Δ^2 term in the above discussion, rather than a single Δ term. A good description of secret sharing schemes for general access structures, including some relatively efficient constructions are presented in the relevant chapter in Stinson's book.

A. Shamir. *How to share a secret.* Communications of the ACM, **22**, 612–613, 1979.

V. Shoup. *Practical threshold signatures.* In Advances in Cryptology – EuroCrypt 2000, Springer-Verlag LNCS 1807, 207–220, 2000.

D. Stinson. *Cryptography: Theory and Practice.* CRC Press, 1995.

CHAPTER 24

Commitments and Oblivious Transfer

Chapter Goals

- To present two protocols which are carried out between mutually untrusting parties.
- To introduce commitment schemes and give simple examples of efficient implementations.
- To introduce oblivious transfer, and again give simple examples of how this can be performed in practice.

1. Introduction

In this chapter we shall examine a number of more advanced cryptographic protocols which enable higher level services to be created. We shall particularly focus on protocols for

- commitment schemes,
- oblivious transfer.

Whilst there is a large body of literature on these protocols, we shall keep our feet on the ground and focus on protocols which can be used in real life to achieve practical higher level services. It turns out that these two primitives are in some sense the most basic atomic cryptographic primitives which one can construct.

Up until now we have looked at cryptographic schemes and protocols in which the protocol participants have been honest, and we are trying to protect their interests against an external adversary. However, in the real world often we need to interact with people who we do not necessarily trust. In this chapter we examine two types of protocol which are executed between two parties, for which each party may want to cheat in some way. In later chapters we shall present more complicated examples of similar protocols. The simplistic protocols in this chapter will form the building blocks on which more complicated protocols can be built.

We start by focusing on commitment schemes, and then we pass to oblivious transfer.

2. Commitment Schemes

Suppose Alice wishes to play 'paper-scissors-stone' over the telephone with Bob. The idea of this game is that Alice and Bob both choose simultaneously one of the set {`paper`,`scissors`,`stone`}. Then the outcome of the game is determined by the rules

- `Paper` wraps `stone`. Hence if Alice chooses `paper` and Bob chooses `stone` then Alice wins.
- `Stone` blunts `scissors`. Hence if Alice chooses `stone` and Bob chooses `scissors` then Alice wins.
- `Scissors` cut `paper`. Hence if Alice chooses `scissors` and Bob chooses `paper` then Alice wins.

If both Alice and Bob choose the same item then the game is declared a draw. When conducted over the telephone we have the problem that whoever goes first is going to lose the game.

One way around this is for the party who goes first to 'commit' to their choice, in such a way that the other party cannot determine what was committed to. Then the two parties can reveal

their choices, with the idea that the other party can then verify that the revealing party has not altered its choice between the commitment and the revealing stage. Such a system is called a commitment scheme. An easy way to do this is to use a cryptographic hash function as follows:

$$\begin{aligned} A &\longrightarrow B : h_A = H(R_A \| \texttt{paper}), \\ B &\longrightarrow A : \texttt{scissors}, \\ A &\longrightarrow B : R_A, \texttt{paper}. \end{aligned}$$

At the end of the protocol Bob needs to verify that the h_A sent by Alice is equal to $H(R_A\|\texttt{paper})$. If the values agree he knows that Alice has not cheated. The result of this protocol is that Alice loses the game since `scissors` cut `paper`.

Let us look at the above from Alice's perspective. She first commits to the value `paper` by sending Bob the hash value h_A. This means that Bob will not be able to determine that Alice has committed to the value `paper`, since Bob does not know the random value of R_A used and Bob is unable to invert the hash function. The fact that Bob cannot determine what value was committed to is called the concealing, or hiding property of a commitment scheme.

As soon as Bob sends the value `scissors` to Alice, she knows she has lost but is unable to cheat, since to cheat she would need to come up with a different value of R_A, say R'_A, which satisfied

$$H(R_A\|\texttt{paper}) = H(R'_A\|\texttt{stone}).$$

But this would mean that Alice could find collisions in the hash function, which for a suitable chosen hash function is believed to be impossible. Actually we require that the hash function is second-preimage resistant in this case. This property of the commitment scheme, that Alice cannot change her mind after the commitment procedure, is called binding.

Let us now study these properties of concealing and binding in more detail. Recall that an encryption function has information theoretic security if an adversary with infinite computing power could not break the scheme, whilst an encryption function is called computationally secure if it is only secure when faced with an adversary with polynomially bounded computing power. A similar division can be made with commitment schemes, but now we have two security properties, namely concealing and binding. One property protects the interests of the sender, and one property protects the interests of the receiver. To simplify our exposition we shall denote our our abstract commitment scheme by a public algorithm, $c = C(x, r)$ which takes a value x and some randomness r and produces a commitment c. To confirm a decomitment the commiter simply reveals the values of x and r. The receiver then checks that the two values produce the original commitment.

DEFINITION 24.1 (Binding). *A commitment scheme is said to be information theoretically (resp. computationally)* binding *if no infinitely powerful (resp. computationally bounded) adversary can win the following game.*

- *The adversary outputs a value c, plus values x and r which produce this commitment.*
- *The adversary must then output a value $x' \neq x$ and a value r' such that*

$$C(x, r) = C(x', r').$$

DEFINITION 24.2 (Concealing). *A commitment scheme is said to be information theoretically (resp. computationally)* concealing *if no infinitely powerful (resp. computationally bounded) adversary can win the following game.*

- *The adversary outputs two messages x_0 and x_1 of equal length.*
- *The challenger generates r at random and a random bit $b \in \{0, 1\}$.*
- *The challenger computes $c = C(x_b, r)$ and passes c to the adversary.*
- *The adversaries goal is to now guess the bit b.*

Notice, how this definition of concealing is virtually identical to our definition of indistinguishability of encryptions. A number of results trivially follow from these two definitions:

LEMMA 24.3. *There exists no scheme which is both information theoretically concealing and binding.*

PROOF. To be perfectly binding a scheme must be deterministic, since there needs to be a one-to-one relationship between the space of commitments and the space of committed values. But a deterministic scheme clearly will not meet the concealing definition. □

LEMMA 24.4. *Using the commitment scheme defined as*

$$H(R\|C),$$

for a random value R, the committed value C and some cryptographic hash function H, is at best

- *computationally binding,*
- *information theoretically concealing.*

PROOF. All cryptographic hash functions we have met are only computationally secure against preimage resistance and second-preimage resistance.

The binding property of the above scheme is only guaranteed by the second-preimage resistance of the underlying hash function. Hence, the binding property is only computationally secure.

The concealing property of the above scheme is only guaranteed by the preimage resistance of the underlying hash function. Hence, the concealing property looks like it should be only computationally secure. However, if we assume that the value R is chosen from a suitably large set, then the fact that the hash function should have many collisions works in our favour and in practice we should obtain something close to information theoretic concealing. On the other hand if we assume that H is a random oracle, then the commitment scheme is clearly information theoretically concealing. □

We now turn to three practical commitment schemes which occur in various protocols. All are based on a finite abelian group G of prime order q, which is generated by g. We let $h \in \langle g \rangle$, where the discrete logarithm of h to the base g is unknown by any user in the system. This latter property is quite easy to ensure, for example for a finite field $\mathbb{F}_p^*$, with q dividing $p-1$ we create g as follows (with a similar procedure being used to determine h):

- Pick a random $r \in \mathbb{Z}$.
- Compute $f = H(r) \in \mathbb{F}_p^*$ for some cryptographic hash function H.
- Set $g = f^{(p-1)/q} \pmod p$. If $g = 1$ then return to the first stage, else output (r, g).

This generates a random element of the subgroup of $\mathbb{F}_p^*$ of order q, with the property that it is generated verifiably at random since one outputs the seed r used to generate the random element.

Given g, h we define two commitment schemes, $B(x)$ and $B_a(x)$, to commit to an integer x modulo q, and one $E_a(x)$ to commit to an integer x modulo p.

$$\begin{aligned} B(x) &= g^x, \\ E_a(x) &= (g^a, x \cdot h^a), \\ B_a(x) &= h^x g^a, \end{aligned}$$

where a is a random integer modulo q. The scheme given by $B_a(x)$ is called Pedersen's commitment scheme. The value a is called the blinding number, since it blinds the value of the commitment x even to a computationally unbounded adversary. To reveal the commitments the user publishes the value x in the first scheme and the pair (a, x) in the second and third schemes.

LEMMA 24.5. *The commitment scheme $B(x)$ is information theoretically binding.*

PROOF. Suppose Alice having published

$$c = B(x) = g^x$$

wished to change her mind as to which element of $\mathbb{Z}/q\mathbb{Z}$ she wants to commit to. Alas, for Alice no matter how much computing power she has there is mathematically only one element in $\mathbb{Z}/q\mathbb{Z}$, namely x, which is the discrete logarithm of the commitment c to the base g. Hence, the scheme is clearly information theoretically binding. □

Note the Pederson commitment scheme does not meet our strong definition of security for the concealing property. If the space of values from which x is selected is large, then this commitment scheme could meet a weaker security definition related to a one-way like property.

LEMMA 24.6. *The commitment scheme $E_a(x)$ is information theoretically binding and computationally concealing.*

PROOF. This scheme is exactly ElGamal encryption with respect to a public key h. Note that we do not need to know the associated private key to use this as a commitment scheme. Indeed any semantically secure public key encryption scheme can be used in this way as a commitment scheme.

The underlying semantic security implies that the resulting commitment scheme is computationally concealing. Whilst the fact that the decryption is unique, implies that the commitment scheme is information theoretically binding. □

LEMMA 24.7. *The commitment scheme $B_a(x)$ is computationally binding and information theoretically concealing. That it is computationally binding only holds if the commiter does not know the discrete logarithm of h to the base g.*

PROOF. Now suppose Alice, after having committed to

$$b = B_a(x) = h^x g^a$$

wishes to change her mind, so as to commit to y instead. All that Alice need do is to compute

$$f = \frac{b}{h^y}.$$

Alice then computes the discrete logarithm a' of f to the base g. When Alice is now asked to reveal her commitment she outputs (a', y) instead of (a, x). Hence the scheme is at most computationally binding.

We can also show that if Alice, after having committed to

$$b = B_a(x) = h^x g^a$$

wishes to change her mind, then the only way she can do this is by computing the discrete logarithm of h to the base g. To see this first note that the value she changes her mind, say y, she must be able to decommit to. Hence, she must know the underlying randomness say b. Thus Alice knows x, y, a and b such that

$$h^x g^a = h^y g^b.$$

From which Alice can recover the discrete logarithm of h with respect to g in the standard manner.

Now suppose the recipient wishes to determine which is the committed value, before the revealing stage is carried out. Since, for a given value of b and every value of x, there is a value of a which makes a valid commitment, even a computationally unbounded adversary could determine no information. Hence, the scheme is information theoretically concealing. □

We end this section by noticing that the two discrete logarithm based commitment schemes we have given possess the homomorphic property:

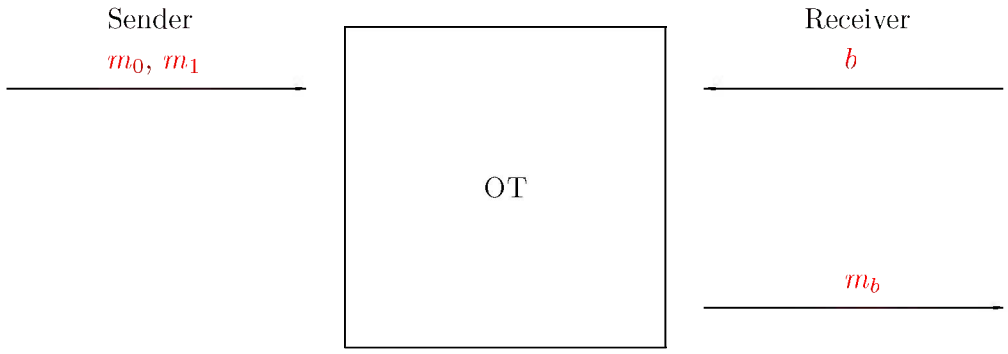
$$\begin{aligned} B(x_1) \cdot B(x_2) &= g^{x_1} \cdot g^{x_2} \\ &= g^{x_1+x_2} \\ &= B(x_1 + x_2), \\ B_{a_1}(x_1) \cdot B_{a_2}(x_2) &= h^{x_1} \cdot g^{a_1} \cdot h^{x_2} \cdot g^{a_2} \\ &= h^{x_1+x_2} \cdot g^{a_1+a_2} \\ &= B_{a_1+a_2}(x_1 + x_2). \end{aligned}$$

We shall use this homomorphic property when we discuss our voting protocol at the end of chapter 25.

3. Oblivious Transfer

We now consider another type of basic protocol called oblivious transfer, or OT for short. This is another protocol which is run between two distrusting parties, a sender and a receiver. In its most basic form the sender has two secret messages as input m_0 and m_1; the receiver has as input a single bit b. The goal of an OT protocol is that at the end of the protocol the sender should not learn the value of the receivers input b. However, the receiver should learn the value of m_b but should learn nothing about m_{1-b}. Such a protocol is often called a 1-out-of-2 OT, since the receiver learns one of the two inputs of the sender. Such a protocol can be visually pictured as in Figure 1. One can easily generalise this concept to an k-out-of-n OT, but as it is we will only be interested in the simpler case.

FIGURE 1. Pictorial Description of an 1-out-of-2 OT



We present a scheme which allows us to perform a 1-out-of-2 oblivious transfer of two arbitrary bit strings m_0, m_1 of equal length. The scheme is based on the following version of ElGamal (resp. DHIES). We take standard discrete logarithm based public/private key pair $(h = g^x, x)$, where g is a generator of cyclic finite abelian group G of prime order q. We will require a hash function H from G to bit strings of length n. Then to encrypt messages m of length n we compute, for a random $k \in \mathbb{Z}_q$

$$c = (c_1, c_2) = \left(g^k, m \oplus H(h^k)\right).$$

To decrypt we compute

$$c_2 \oplus H(c_1{}^x) = m \oplus H(h^k) \oplus H(g^{kx}) = m.$$

Notice, that the first part of the ciphertext corresponds to DHIES KEM, but the second part corresponds to a CPA secure DEM. It can be shown that the above scheme is semantically secure under chosen plaintext attacks (i.e. passive attacks) in the random oracle model.

The idea behind our oblivious transfer protocol is for the receiver to create two public keys h_0 and h_1, only one of which it knows the corresponding secret key for. If the receiver knows the secret key for h_b, where b is the bit he is choosing, then he can decrypt for messages encrypted under this key, but not decrypt under the other key. The sender then only needs to encrypt his messages with the two keys. Since the receiver only knows one secret key he can only decrypt one of the message.

To implement this idea concretely, the sender first selects a random element c in G, it is important that the receiver does not know the discrete logarithm of c with respect to g. This value is then sent to the sender. The sender then generates two public keys, according to his bit b, via first generating $x \in \mathbb{Z}_q$ and then computing

$$h_b = g^x, \;\; h_{1-b} = c/h_b.$$

Notice, that the receiver knows the underlying secret key for h_b, but he does not know the secret key for h_{1-b} since he does not know the discrete logarithm of c with respect to g. These two public key values are then sent to the sender. The sender then encrypts message m_0 using the key h_0 and message m_1 using key h_1, i.e. the sender computes

$$\begin{aligned} e_0 &= \left(g^{k_0}, m_0 \oplus H({h_0}^{k_0})\right), \\ e_1 &= \left(g^{k_1}, m_1 \oplus H({h_1}^{k_1})\right), \end{aligned}$$

for two random integers k_0, $k_1 \in \mathbb{Z}_q$. These two ciphertexts are then sent to the receiver who then decrypts the bth one using his secret key x.

From the above description we can obtain some simple optimisations. Firstly, the receiver does not need to send both h_0 and h_1 to the sender, since the sender can always compute h_1 from h_0 by computing c/h_0. Secondly, we can use the same value of $k = k_0 = k_1$ in the two encryptions. We thus obtain the following oblivious transfer protocol

Sender		Receiver
$c \in G$	$\xrightarrow{c}$	
		$x \in \mathbb{Z}_q$,
		$h_b = g^x$,
	$\xleftarrow{h_0}$	$h_{1-b} = c/h_b$,
$h_1 = c/h_0$		
$k \in \mathbb{Z}_q$		
$c_1 = g^k$		
$e_0 = m_0 \oplus H({h_0}^k)$		
$e_1 = m_1 \oplus H({h_1}^k)$	$\xrightarrow{c_1, e_0, e_1}$	
		$m_b = e_b \oplus H({c_1}^x)$.

So does this respect the two conflicting security requirements of participants? First, note that the sender cannot determine the hidden bit b of the receiver since the value h_0 sent from the receiver is simply a random element in G. Then we note that the receiver can learn nothing about m_{1-b} since to do this they would have to be able to compute the output of H on the value h_{1-b}^k, which would imply contradicting the fact that H acts as a random oracle or being able to solve the Diffie–Hellman problem in the group G.

Chapter Summary

- We introduced the idea of protocols between mutually untrusting parties, and introduced commitment and oblivious transfer as two simple examples of such protocols.
- A commitment scheme allows one party to bind themselves to a value, and then reveal it later.
- A commitment scheme needs to be both binding and concealing. Efficient schemes exist which are either information theoretically binding or information theoretically concealing, but not both.
- An oblivious transfer protocol allows a sender to send one of two messages to a recipient, but he does not know which message is actually obtained. The receiver also learns nothing about the other message which was sent.

Further Reading

The above oblivious transfer protocol originally appeared, in a slightly modified form in the paper by Bellare and Micali. The paper by Naor and Pinkas discusses a number of optimisations of the oblivious transfer protocol which we presented above. In particular it presents mechanisms to perform efficiently 1-out-of-N oblivious transfer.

M. Bellare and S. Micali. *Non-interactive oblivious transfer and applications.* In Advances in Cryptology – Crypto '89, Springer-Verlag LNCS 435, 547–557, 1990.

M. Naor and B. Pinkas. *Efficient oblivious transfer protocols.* In SIAM Symposium on Discrete Algorithms – SODA 2001.

CHAPTER 25

Zero-Knowledge Proofs

Chapter Goals

- To introduce zero-knowledge proofs.
- To explain the notion of simulation.
- To introduce Sigma protocols.
- To explain how these can be used in a voting protocol.

1. Showing a Graph Isomorphism in Zero-Knowledge

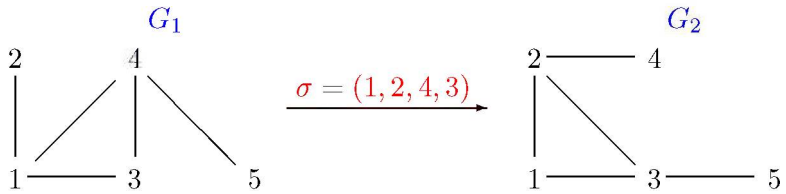
Suppose Alice wants to convince Bob that she knows something without Bob finding out exactly what Alice knows. This apparently contradictory state of affairs is dealt with using zero-knowledge proofs. In the literature of zero-knowledge proofs the role of Alice is called the prover, since she wishes to prove something, whilst the role of Bob is called the verifier, since he wishes to verify that the prover actually knows something. Often, and we shall also follow this convention, the prover is called Peggy and the verifier is called Victor.

The classic example of zero-knowledge proofs is based on the graph isomorphism problem. Given two graphs G_1 and G_2, with the same number of vertices, we say that the two graphs are isomorphic if there is a relabelling (i.e. a permutation) of the vertices of one graph which produces the second graph. This relabelling ϕ is called the graph isomorphism which is denoted by

$$\phi : G_1 \longrightarrow G_2.$$

It is a hard computational problem to determine a graph isomorphism between two graphs. As a running example consider the two graphs in Figure 1, linked by the permutation $\phi = (1, 2, 4, 3)$.

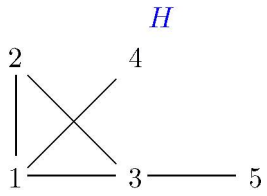
FIGURE 1. Example graph isomorphism



Suppose Peggy knows the graph isomorphism ϕ between two public graphs G_1 and G_2. We call ϕ the prover's private input, whilst the groups G_1 and G_2 are the public or common input. Peggy wishes to convince Victor that she knows the graph isomorphism, without revealing to Victor the precise nature of the graph isomorphism. This is done using the following zero-knowledge proof.

Peggy takes the graph G_2 and applies a secret random permutation ψ to the vertices of G_2 to produce another isomorphic graph H. In our running example we take $\psi = (1, 2)$, the isomorphic graph H is then given by Figure 2.

FIGURE 2. Peggy's committed graph



Peggy now publishes H as a **commitment**, she of course knows the following secret graph isomorphisms

$$\begin{aligned} \phi &: G_1 \longrightarrow G_2, \\ \psi &: G_2 \longrightarrow H, \\ \psi \circ \phi = \phi \cdot \psi &: G_1 \longrightarrow H. \end{aligned}$$

Victor now gives Peggy a **challenge**, he selects $b \in \{1, 2\}$ and asks for the graph isomorphism between H and G_b. Peggy now gives her **response** by returning either $\chi = \psi$ or $\chi = \phi \cdot \psi$. The transcript of the protocol then looks like

$$\begin{aligned} P &\longrightarrow V : H, \\ V &\longrightarrow P : b, \\ P &\longrightarrow V : \chi. \end{aligned}$$

In our example if Victor chooses $b = 2$ then Peggy simply needs to publish ψ. However, if Victor chooses $b = 1$ then Peggy publishes

$$\phi \cdot \psi = (1, 2, 4, 3) \cdot (1, 2) = (2, 4, 3).$$

We can then see that $(2, 4, 3)$ is the permutation which maps graph G_1 onto graph H. But to compute this we needed to know the hidden isomorphism ϕ.

Let us first examine how Peggy can cheat. If Peggy does not know the graph isomorphism ϕ then she will need to know, before Victor gives his challenge, the graph G_b which Victor is going to pick. Hence, if Peggy is cheating she will only be able to respond to Victor correctly 50 percent of the time. So, repeating the above protocol a number of times, a non-cheating Peggy will be able to convince Victor that she really does know the graph isomorphism, with a small probability of error of Victor being convinced incorrectly.

Now we need to determine whether Victor learns anything from running the protocol, i.e. is Peggy's proof really zero-knowledge? We first notice that Peggy needs to produce a different value of H on every run of the protocol, otherwise Victor can trivially cheat. We assume therefore that this does not happen.

One way to see whether Victor has learnt something after running the protocol is to look at the transcript of the protocol and ask after having seen the transcript whether Victor has gained any knowledge, or for that matter whether anyone looking at the protocol but not interacting learns anything. One way to see that Victor has not learnt anything is to see that Victor could have written down a valid protocol transcript without interacting with Peggy at all. Hence, Victor cannot use the protocol transcript to convince someone else that he knows Peggy's secret isomorphism. He cannot even use the protocol transcript to convince another party that Peggy knows the secret graph isomorphism.

Victor can produce a valid protocol transcript using the following simulation:

- Choose $b \in \{1, 2\}$.
- Generate a random isomorphism χ of the graph G_b to produce the graph H.

- Output the transcript

$$P \longrightarrow V : H,$$
$$V \longrightarrow P : b,$$
$$P \longrightarrow V : \chi.$$

Hence, the interactive nature of the protocol means that it is a zero-knowledge proof. We remark that the three-pass system of

$$\text{commitment} \longrightarrow \text{challenge} \longrightarrow \text{response}$$

is the usual characteristic of such protocols.

Just as with commitment schemes we can divide zero-knowledge protocols into categories depending on whether they are secure with respect to computationally bounded or unbounded adversaries. Clearly two basic properties of an interactive proof system are

- **Completeness:** If Peggy really knows the thing being proved, then Victor should accept her proof with probability one.
- **Soundness:** If Peggy does not know the thing being proved, then Victor should only have a small probability of actually accepting the proof.

We usually assume that Victor is a polynomially bounded party, whilst Peggy is unbounded. In the above protocol based on graph isomorphism we saw that the soundness probability was equal to one half. Hence, we needed to repeat the protocol a number of times to reduce this to something small.

The zero-knowledge property we have already noted is related to the concept of a simulation. Suppose the set of valid transcripts (produced by true protocol runs) is denoted by $\mathcal{V}$ and let the set of possible simulations be denoted by $\mathcal{S}$. The security is therefore related to how much like the set $\mathcal{V}$ is the set $\mathcal{S}$.

A zero-knowledge proof is said to have perfect zero-knowledge if the two sets $\mathcal{V}$ and $\mathcal{S}$ cannot be distinguished from each other by a computationally unbounded adversary. However, if the two sets are only indistinguishable by a computationally bounded adversary we say that the zero-knowledge proof has computational zero-knowledge.

2. Zero-Knowledge and $\mathcal{NP}$

So the question arises as to what can be shown in zero-knowledge. Above we showed that the knowledge of whether two graphs are isomorphic can be shown in zero-knowledge. Thus the decision problem of `Graph Isomorphism` lies in the set of all decision problems which can be proven in zero-knowledge. But `Graph Ismorphism` is believed to lie between the complexity classes $\mathcal{P}$ and $\mathcal{NP}$-complete, i.e. it can neither be solved in polynomial time, yet neither is it $\mathcal{NP}$-complete.

We can think of $\mathcal{NP}$ problems as those problems for which there is a witness (or proof) which can be produced by an all powerful prover, but which a polynomially bounded verifier can verify the proof. However, for the class of $\mathcal{NP}$ problems the prover and the verifier do not interact, i.e. the proof is produced and then the verifier verifies it.

If we allow interation then something quite amazing happens. Consider an all powerful prover who interacts with a polynomially bounded verifier. We wish the prover to convince the verifier of the validity of some statement. This is exactly as we had in the previous section except that we only require the completeness and soundness properties, i.e. we do not require the zero-knowledge property. The decision problems which can be proved to be true in such a manner form the complexity class of *interactive proofs*, or $\mathcal{IP}$. It can be shown that the complexity class $\mathcal{IP}$ is equal to the complexity class $\mathcal{PSPACE}$, i.e. the set of all decision problems which can be solved using

polynomial space. It is widely believed that $\mathcal{NP} \subsetneq \mathcal{PSPACE}$, which implies that having interaction really gives us something extra.

So what happens to interactive proofs when we add in the zero-knowledge requirement? We can define a complexity class $\mathcal{CZK}$ of all decision problems which can be verified to be true using a computational zero-knowledge proof. We have already shown that the problem of `Graph Isomorhism` lies in $\mathcal{CZK}$, but this might not include all of the $\mathcal{NP}$ problems.

However, since 3-colourability is $\mathcal{NP}$-complete, we have the following elegant proof that $\mathcal{NP} \subset \mathcal{CZK}$,

THEOREM 25.1. *The problem of 3-colourability of a graph lies in $\mathcal{CZK}$, assuming a computationally hiding commitment scheme exists.*

PROOF. Consider a graph $G = (V, E)$ in which the prover knows a colouring ψ of the G, i.e. a map $\psi : V \to \{1, 2, 3\}$ such that $\psi(v_1) \neq \psi(v_2)$ if $(v_1, v_2) \in E$. The prover first selects a commitment scheme $C(x; r)$ and a random permutation π of the set $\{1, 2, 3\}$. Note, the function $\pi(\psi(v))$ defines another three colouring of the graph. Now the prover commits to this second three colouring by sending to the verifier the commitments

$$c_i = C\left(\pi(\psi(v_i)); r_i\right) \text{ for all } v_i \in V.$$

The verifier then selects a random edge $(v_i, v_j) \in E$ and sends this to the prover. The prover now decommits to the values of

$$\pi(\psi(v_i)) \text{ and } \pi(\psi(v_j)),$$

and the verifier checks that

$$\pi(\psi(v_i)) \neq \pi(\psi(v_j)).$$

We now turn to the three required properties of a zero-knowledge proof.

Completeness: The above protocol is complete since any valid prover will get the verifier to accept with probability one.
Soundness: If we have a cheating prover then at least one edge is invalid, and with probability at least $1/|E|$ the verifier will select an invalid edge. Thus with probability at most $1 - 1/|E|$ a cheating prover will get a verifier to accept. By repeating the above proof many times one can reduce this probability to as low a value as we require.
Zero-Knowledge: Assuming the commitment scheme is computationally hiding the obvious simulation and the real protocol will be computationally indistinguishable. □

Notice, that this is a very powerful result. It says that virtually any statement which is like to come up in cryptography can be proved in zero-knowledge. Clearly the above proof would not provide a practical implementation, but at least we know that very powerful tools can be applied. In the next section we turn to more practical proofs which can be applied in practice. But before doing that we note that the above result can be extended even further

THEOREM 25.2. *Assuming one-way functions exist then $\mathcal{CZK} = \mathcal{IP}$, and hence $\mathcal{CZK} = \mathcal{PSPACE}$.*

3. Sigma Protocols

One can use a zero-knowledge proof of possession of some secret as an identification scheme. The trouble with the above protocol for graph isomorphisms is that it is not very practical. The data structures required are very large, and the protocol needs to be repeated a large number of times before Victor is convinced that Peggy really knows the secret.

However one can do even better in many instances if one restricts to so called *honest-verifier* zero-knowledge protocols. In the case of honest-verifier's one assumes that the verifier responds randomly to the commitment. In other words the verifier's challenge is uniformly distributed over

the range of possible challenges and does not depend on the value of the prior commitment. A three round honest-verifier zero-knowledge protocol is often called a Sigma protocol.

For Sigma protocols we can use a different form of soundness called special-soundness, which is often easier to use in practice. A three round protocol as above is said to have the special-soundness property if given two protocol runs with the same commitment but different challenges one can recover the provers secret. We note that in the case of honest-verifiers the special-soundness property imples the usual soundness property.

The previous zero knowledge protocol based on graph isomorphism was very inefficient, in that it needed to be repeated a large number of times to reduce the soundness probability to something suitably small. However, if we restrict to Sigma protocols we can be more efficient.

3.1. Schnorr's Identification Protocol. In essence we have already seen a Sigma protocol which has better bandwidth and error properties when we discussed Schnorr signatures in Chapter 14. Suppose Peggy's secret is now the discrete logarithm x of y with respect to g in some finite abelian group G of prime order q.

The protocol for proof of knowledge now goes as follows

$$\begin{aligned} &P \longrightarrow V : r = g^k \text{ for a random } k, \\ &V \longrightarrow P : e, \\ &P \longrightarrow V : s = k + xe \pmod{q}. \end{aligned}$$

Victor now verifies that Peggy knows the secret discrete logarithm x by verifying that

$$r = g^s y^{-e}.$$

Let us examine this protocol in more detail. We first note that the protocol is complete, in that if Peggy actually knows the discrete logarithm then Victor will accept the protocol.

We now turn to soundness, if Peggy does not know the discrete logarithm x then one can informly argue that she will only be able to cheat with probability $1/q$, which is much better than the $1/2$ from the earlier graph isomorphism based protocol. We can however show that the protocol has the special-soundness property; Suppose that we have two protocol runs with transcripts

$$(r, e, s) \text{ and } (r, e', s').$$

Note, that the commitments are equal but that the challenges (and hence responses) are different. We need to show that we can from this data recover x. As the protocol verifies this implies that we must have

$$r = g^s y^{-e} = g^{s'} y^{-e'} = r.$$

Which implies

$$s + x(-e) = s' + x(-e') \pmod{q}.$$

Hence, we recover x via

$$x = \frac{s - s'}{e - e'} \pmod{q}.$$

Notice, that this proof of soundness is almost exactly the same as our argument via the forking lemma that Schnorr signatures are secure with respect to passive adversaries in the random oracle model, Theorem 20.1.

But does Victor learn anything from the protocol? The answer to this is no, since Victor could simulate the whole transcript in the following way.

- Generate a random value of e modulo q.
- Compute $r = g^s y^{-e}$.

- Output the transcript

$$P \longrightarrow V : r,$$
$$V \longrightarrow P : e,$$
$$P \longrightarrow V : s.$$

In other words the protocol is zero-knowledge, in that someone cannot tell the simulation of a transcript from a real transcript. This is exactly the same simulation we used when showing that Schnorr signature are secure against active adversaries in Section 2.2.1.

One problem with the above Sigma protocols is that they are interactive in nature:

$$P \longrightarrow V : \text{Co},$$
$$V \longrightarrow P : \text{Ch},$$
$$P \longrightarrow V : \text{Re},$$

where Co is the commitment, Ch is the challenge and Re is the response. They can easily be made non-interactive by replacing the challenge with the evaluation of a cryptographic hash function applied to the commitment,

$$\text{Ch} = H(\text{Co}).$$

The idea here is that the prover cannot fix the challenge before coming up with the commitment, since that would imply they could invert the hash function. The non-interactive protocol no longer has the zero-knowledge property, since it is not simulatable. In addition a computationally unbounded Peggy could invert the hash function, hence we need to restrict the prover to be computationally bounded. In such a situation, where we restrict to a computationally bounded prover, we say we have a zero-knowledge argument as opposed to a zero-knowledge proof.

One could also add some other data into the value which is hashed, for example a message. In this way we turn an interactive Sigma protocol into a digital signature scheme,

$$\text{Ch} = H(\text{Co} \| \text{Message}).$$

You should now be able to see how Schnorr signatures are exactly what one obtains when one uses the hash of the commitment and a message as the challenge, in the above proof of knowledge of discrete logarithms. This transformation, of Sigma protocols into signature schemes, is called the Fiat–Shamir heuristic.

Before we discuss other Sigma protocols we introduce some notation, to aid our discussion. Suppose we wish to prove knowledge of the variable x via a Sigma protocol, we then let

- $R(x, k)$ denote the algorithm used to compute the commitmemt r, where k is a random nonce.
- c is the challenge.
- $S(c, x, k)$ denote the algorithm which the prover uses to compute their response s response.
- $V(r, c, s)$ will denote the verification algorithm.
- $S'(c, s)$ denotes the simulators algorithm which creates a value of a commitment r which will verify the transcript (r, c, s).

All algorithms are assumed to implicitly have as input the public value which x is linked to.

Using this notation Schnorr's identification protocol becomes the following: The variable we wish to prove knowledge of is x where $y = g^x$. We then have

$$\begin{aligned} R(x, k) &= r = g^k, \\ S(c, x, k) &= s = k + c \cdot x \pmod{q}, \\ V(r, c, s) &= \textbf{true} \Leftrightarrow (g^s = r \cdot y^{-c}), \\ S'(c, s) &= r = g^s \cdot y^c. \end{aligned}$$

3.2. Chaum–Pedersen Protocol. We now present a Sigma protocol called the Chaum–Pedersen protocol which was first presented in the context of electronic cash systems, but which has very wide application.

Suppose Peggy wishes to prove she knows two discrete logarithms

$$y_1 = g^{x_1} \text{ and } y_2 = h^{x_2}$$

such that $x_1 = x_2$, i.e. we wish to present both a proof of knowledge of the discrete logarithm, but also a proof of equality of the hidden discrete logarithms. We assume that g and h generate groups of prime order q, and we denote the common discrete logarithm by x ease notation. Using our prior notation for Sigma protocols, the Chaum–Pedersen protocol can be expressed via

$$\begin{aligned} R(x,k) &= (r_1,r_2) = (g^k, h^k), \\ S(c,x,k) &= s = k - c\cdot x \pmod q, \\ V((r_1,r_2),c,s) &= \textbf{true} \Leftrightarrow (r_1 = g^s \cdot y_1^c \text{ and } r_2 = h^s \cdot y_2^c), \\ S'(c,s) &= (r_1,r_2) = (g^s \cdot y_1^c, h^s \cdot y_2^c). \end{aligned}$$

Note, how this resembles two concurrent runs of the Schnorr protocol.

The Chaum–Pedersen protocol is clearly both complete and has the zero-knowledge property, the second fact follows since the simulation $S'(c,s)$ produces transcripts which are is indistinguishable from a real transcript. We need to show it is sound, however since we are assuming honest-verifiers we only need to show it has the special-soundness property. Hence, we assume two protocol runs with the same commitments (t_1,t_2), different challenges, c_1 and c_2 and valid responses s_1 and s_2. With this data we need to show that this reveals the common discrete logarithm. Since the two transcripts pass the verification test we have such that

$$t_1 = g^{s_1}\cdot y_1^{c_1} = g^{s_2}\cdot y_1^{c_2} \text{ and } t_2 = h^{s_1}\cdot y_2^{c_1} = h^{s_2}\cdot y_2^{c_2}$$

But this implies that

$$y_1^{c_1-c_2} = g^{s_2-s_1} \text{ and } y_2^{c_2-c_1} = h^{s_2-s_1}$$

Hence, the two discrete logarithms are equal and can be extracted from

$$x = \frac{c_1 - c_2}{s_2 - s_1} \pmod q.$$

3.3. Proving Knowledge of Pedersen Commitments. Often one commits to a value using a commitment scheme, but the receiver is not willing to proceed unless one proves one knows the value committed to. In other words the reciever will only proceed if he knows that the sender will at some point be able to reveal the value committed to.

For the commitment scheme

$$B(x) = g^x$$

this is simple, we simply execute Schnorr's protocol for proof of knowledge of a discrete logarithm. For Pedersen commitments

$$B_a(x) = h^x g^a$$

we need something different.

In essence we wish to prove knowledge of x_1 and x_2 such that

$$y = {g_1}^{x_1}\cdot {g_2}^{x_2}$$

where g_1 and g_2 are elements in a group of prime order q. We note that the following protocol generalises easily to the case when we have more bases, i.e.

$$y = {g_1}^{x_1}\cdots {g_n}^{x_n}.$$

A generalisation which we leave as an exercise.

$$\begin{aligned}
R(x,k) &= (k_1,k_2) = (r_1,r_2) = ({g_1}^{k_1},{g_2}^{k_2}),\\
S(c,\{x_1,x_2\},\{k_1,k_2\}) &= (s_1,s_2)\\
&= (k_1 + c\cdot x_1 \pmod q, k_2 + c\cdot x_2 \pmod q),\\
V((r_1,r_2),c,(s_1,s_2)) &= \textbf{true} \Leftrightarrow (g_1^{s_1}\cdot g_2^{s_2} = y^c\cdot t_1\cdot t_2),\\
S'(c,(s_1,s_2)) &= (r_1,r_2)\\
&= \left(\begin{array}{l} r_1 \text{ chosen at random from the group,}\\ g_1^{s_1}\cdot g_2^{s_2}\cdot y^c/r_1 \end{array}\right).
\end{aligned}$$

We leave it to the reader to verify that this protocol is complete, zero-knowledge and satisfies the special-soundness property.

3.4. "Or" Proofs. Sometimes the statement about which we wish to execute a Sigma protocol is not as clear cut as the previous examples. For example suppose we wish to show we know either a secret x or a secret y, without revealing which of the two secrets we know. This is a very common occurance which arises in a number of advanced protocols, including the voting protocol we consider later in this chapter. It turns out that to show knowledge of one thing or another can be performed using an elegant protocol due to Cramer, Damgård and Schoenmakers.

First assume that there already exists a Sigma protocol to prove knowledge of both secrets individually. The idea is to combine these two Sigma protocols together into one protocol which proves the statement we require. The key idea is as follows: For the secret we know we run the Sigma protocol as normal, however for the secret we do not know we run the simulated Sigma protocol. These two protocols are then linked together by linking the commitments.

As a high level example, suppose the protocol for proving knowledge of x is given by the set of algorithms

$$R_1(x,k_1), S_1(c_1,x,k_1), V(r_1,c_1,s_1), S_1'(c_1,s_1).$$

Similarly we let the Sigma protocol to prove knowlegde of y be given by

$$R_2(y,k_2), S_2(c_2,y,k_2), V(r_2,c_2,s_2), S_2'(c_2,s_2).$$

We assume that the challenges c_1 and c_2 are bit strings of the same length in what follows.

Now suppose we know x, but not y, then our algorithms for the combined proof become:

$$\begin{aligned}
R(x,k_1) &= (r_1,r_2)\\
&= \left\{\begin{array}{l} r_1 = R_1(x,k_1)\\ \text{Select } c_2,s_2 \text{ from the correct distributions}\\ r_2 = S_2'(c_2,s_2)\end{array}\right.\\
S(c,x,k_1) &= (c_1,c_2,s_1,s_2)\\
&= \left\{\begin{array}{l} c_1 = c\oplus c_2\\ s_1 = S_1(c_1,x,k_1)\end{array}\right).\\
V((r_1,r_2),c,(c_1,c_2,s_1,s_2) &= \textbf{true} \Leftrightarrow\\
&\quad c = c_1\oplus c_2 \text{ and } V_1(r_1,c_1,s_1) \text{ and } V_2(r_2,c_2,s_2)\\
S'(c,(c_1,c_2,s_1,s_2)) &= (r_1,r_2)\\
&= \big(S_1'(c_1,s_1), S_2'(c_2,s_2)\big)
\end{aligned}$$

Note that the prover does not reveal the value of c_1, c_2 or s_2 until the response stage of the protocol. Also note that in the simulated protocol the correct distributions of c, c_1 and c_2 are such that $c = c_1\oplus c_2$. The protocol for the case where we know y but not x follows by reversing the roles of c_1 and c_2 and r_1 and r_2 in the algorithms R, S and V. If the prover knows both x and y then

they can execute either of the two possibilties. The completeness, soundness and zero-knowledge properties follow from the corresponding properties of the original Sigma protocols.

We now present a simple example which uses the Schnorr protocol as a building block. Suppose we wish to prove knowledge of either x_1 or x_2 such that

$$y_1 = g^{x_1} \text{ and } y_2 = g^{x_2},$$

where g lies in a group G of prime order q. We assume that the prover knows x_i but not x_j where $i \neq j$.

The provers commitment is (r_1, r_2) is computed by selecting c_j and k_i uniformly at random from ${\mathbb{F}_q}^*$ and s_j uniformly at random from G. They then compute

$$r_i = g^{k_i} \text{ and } r_j = g^{s_j} \cdot y_j^{-c_j}.$$

On recieving the challeneg $c \in {\mathbb{F}_q}^*$ the prover computes

$$\begin{aligned} c_i &= c + c_j \pmod{q} \\ s_i &= k_i + c_i \cdot x_i \pmod{q}. \end{aligned}$$

Note, we have replaced $\oplus$ in computing the "challenge" c_i into addition modulo q, a moments thought reveals that this is a better way at preserving the relative distributions in this example. The prover then outputs (c_1, c_2, s_1, s_2)

The verifier checks the proof by checking that

$$c = c_1 + c_2 \pmod{q} \text{ and } r_1 = g^{s_1} \cdot y_1^{-c_1} \text{ and } r_2 = g^{s_2} \cdot y_2^{-c_2}.$$

These "Or" proofs can be extended to an arbitrary number of disjunctions of statements in the obvious manner: Given n statements of which the prover only knows one secret,

- Simulate $n-1$ statements using the simulations and challenges c_i
- Commit as usual to the known statement
- Generate correct challenge for the known statement via

$$c = c_1 \oplus \cdots \oplus c_n$$

3.5. A more complicated example. We end this section by giving a protocol, which is a zero-knowledge argument, which will be required when we discuss voting schemes. It is obtained by combining the protocol for proving knowledge of Pedersen commitments with the "or" proofs of the prior section.

Consider the earlier commitment scheme given by

$$B_a(x) = h^x g^a,$$

where $G = \langle g \rangle$ is a finite abelian group of prime order q, h is an element of G whose discrete logarithm with respect to g is unknown, x is the value being committed and a is a random nonce. We are interested in the case where the value committed is restricted to be either plus or minus one, i.e. $x \in \{-1, 1\}$. It will be important in our application for the person committing to prove that their commitment is from the set $\{-1, 1\}$ without revealing what the actual value of the commitment is.

To do this we execute the following protocol.

- As well as publishing the commitment $B_a(x)$, Peggy also chooses random numbers d, r and w modulo q and then publishes α_1 and α_2 where

$$\alpha_1 = \begin{cases} g^r \left(B_a(x)h\right)^{-d} & \text{if } x = 1 \\ g^w & \text{if } x = -1, \end{cases}$$

$$\alpha_2 = \begin{cases} g^w & \text{if } x = 1 \\ g^r \left(B_a(x)h^{-1}\right)^{-d} & \text{if } x = -1. \end{cases}$$

- Victor now sends a random challenge c to Peggy.
- Peggy responds by setting

$$d' = c - d,$$
$$r' = w + ad'.$$

  Then Peggy returns the values

$$(d_1, d_2, r_1, r_2) = \begin{cases} (d, d', r, r') & \text{if } x = 1 \\ (d', d, r', r) & \text{if } x = -1. \end{cases}$$

- Victor then verifies that the following three equations hold

$$c = d_1 + d_2,$$
$$g^{r_1} = \alpha_1 (B_a(x)h)^{d_1},$$
$$g^{r_2} = \alpha_2 (B_a(x)h^{-1})^{d_2}.$$

To show that the above protocol works we need to show that

(1) If Peggy responds honestly then Victor will verify that the above three equations hold.
(2) If Peggy has not committed to plus or minus one then she will find it hard to produce a response to Victor's challenge which is correct.
(3) The protocol reveals no information to any party as to the exact value of Peggy's commitment, bar that it come from the set $\{-1, 1\}$.

We leave the verification of these three points to the reader. Note that the above protocol can clearly be conducted in a non-interactive manner by defining

$$c = H\left(\alpha_1 \| \alpha_2 \| B_a(x)\right).$$

4. An Electronic Voting System

In this section we describe an electronic voting system which utilizes some of the primitives we have been discussing in this chapter and in earlier chapters. In particular we make use of secret sharing schemes from Chapter 23, commitment schemes from Chapter 24, and zero-knowledge proofs from this chapter. The purpose is to show how basic cryptographic primitives can be combined into a complicated application giving real value. One can consider an electronic voting scheme to be a special form of a secure multi-party computation, a topic which we shall return to in Chapter 26.

Our voting system will assume that we have m voters, and that there are n centres which perform the tallying. The use of a multitude of tallying centres is to allow voter anonymity and stop a few centres colluding to fix the vote. We shall assume that voters are only given a choice of one of two candidates, for example Democrat or Republican.

The voting system we shall describe will have the following seven properties.

(1) Only authorized voters will be able to vote.
(2) No one will be able to vote more than once.
(3) No stakeholder will be able to determine how someone else has voted.

(4) No one can duplicate someone else's vote.
(5) The final result will be correctly computed.
(6) All stakeholders will be able to verify that the result was computed correctly.
(7) The protocol will work even in the presence of some bad parties.

4.1. System Setup. Each of the n tally centres has a public key encryption function E_i. We assume a finite abelian group G is fixed, of prime order q, and two elements $g, h \in G$ are selected for which no party (including the tally centres) know the discrete logarithm

$$h = g^x.$$

Each voter has a public key signature algorithm.

4.2. Vote Casting. Each of the m voters picks a vote v_j from the set $\{-1, 1\}$. The voter picks a random blinding value $a_j \in \mathbb{Z}/q\mathbb{Z}$ and publishes their vote

$$B_j = B_{a_j}(v_j),$$

using the bit commitment scheme given earlier. This vote is public to all participating parties, both tally centres and other voters. Along with the vote B_j the voter also publishes a non-interactive version of the earlier protocol to show that the vote was chosen from the set $\{-1, 1\}$. The vote and its proof are then digitally signed using the signing algorithm of the voter.

4.3. Vote Distribution. We now need to distribute the votes cast around the tally centres so that the final tally can be computed. Each voter employs Shamir secret sharing as follows, to share the a_j and v_j around the tallying centres: Each voter picks two random polynomials modulo q of degree $t < n$.

$$R_j(X) = v_j + r_{1,j}X + \cdots + r_{t,j}X^t,$$
$$S_j(X) = a_j + s_{1,j}X + \cdots + s_{t,j}X^t.$$

The voter computes

$$(u_{i,j}, w_{i,j}) = (R_j(i), S_j(i)) \text{ for } 1 \le i \le n.$$

The voter encrypts the pair $(u_{i,j}, w_{i,j})$ using the ith tally centre's encryption algorithm E_i. This encrypted share is sent to the relevant tally centre. The voter then publishes it commitments to the polynomial $R_j(X)$ by publicly posting

$$B_{l,j} = B_{s_{l,j}}(r_{l,j}) \text{ for } 1 \le l \le t,$$

again using the earlier commitment scheme.

4.4. Consistency Check. Each centre i needs to check that the values of

$$(u_{i,j}, w_{i,j})$$

it has received from voter j are consistent with the commitments made by the voter. This is done by verifying the following equation

$$\begin{aligned} B_j \prod_{l=1}^{t} B_{l,j}{}^{i^l} &= B_{a_j}(v_j) \prod_{l=1}^{t} B_{s_{l,j}}(r_{l,j})^{i^l} \\ &= h^{v_j} g^{a_j} \prod_{l=1}^{t} (h^{r_{l,j}} g^{s_{l,j}})^{i^l} \\ &= h^{(v_j + \sum_{l=1}^{t} r_{l,j} i^l)} g^{(a_j + \sum_{l=1}^{t} s_{l,j} i^l)} \\ &= h^{u_{i,j}} g^{w_{i,j}}. \end{aligned}$$

4.5. Tally Counting. Each of the n tally centres now computes and publicly posts its sum of the shares of the votes cast

$$T_i = \sum_{j=1}^{m} u_{i,j}$$

plus it posts its sum of shares of the blinding factors

$$A_i = \sum_{j=1}^{m} w_{i,j}.$$

Every other party, both other centres and voters can check that this has been done correctly by verifying that

$$\begin{aligned}\prod_{j=1}^{m}\left(B_j \prod_{l=1}^{t} B_{l,j}^{j^l}\right) &= \prod_{j=1}^{m} h^{u_{i,j}} g^{w_{i,j}} \\ &= h^{T_i} g^{A_i}.\end{aligned}$$

Any party can compute the final tally by taking t of the values T_i and interpolating them to reveal the final tally. This is because T_i is the evaluation at i of a polynomial which shares out the sum of the votes. To see this we have

$$\begin{aligned}T_i &= \sum_{j=1}^{m} u_{i,j} \\ &= \sum_{j=1}^{m} R_j(i) \\ &= \left(\sum_{j=1}^{m} v_j\right) + \left(\sum_{j=1}^{m} r_{1,j}\right) i + \cdots + \left(\sum_{j=1}^{m} r_{t,j}\right) i^t.\end{aligned}$$

If the final tally is negative then the majority of people voted -1, whilst if the final tally is positive then the majority of people voted $+1$. You should now convince yourself that the above protocol has the seven properties we said it would at the beginning.

Chapter Summary

- An interactive proof of knowledge leaks no information if the transcript could be simulated without the need of the secret information.
- Both interactive proofs and zero-knowledge proofs are a very powerful construct, they can be used to prove any statement in $\mathcal{PSPACE}$.
- Interactive proofs of knowledge can be turned into digital signature algorithms by replacing the challenge by the hash of the commitment concatenated with the message.
- Quite complicated protocols can then be built on top of our basic primitives of encryption, signatures, commitment and zero-knowledge proofs. As an example we gave an electronic voting protocol.

Further Reading

The book by Goldreich has more details on zero-knowledge proofs, whilst a good overview of this area is given in Stinson's book. The voting scheme we describe is given in the paper of Cramer et al. from EuroCrypt.

R. Cramer, M. Franklin, B. Schoenmakers and M. Yung. *Multi-authority secret-ballot elections with linear work.* In Advances in Cryptology – EuroCrypt '96, Springer-Verlag LNCS 1070, 72–83, 1996.

O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudo-randomness.* Springer-Verlag, 1999.

D. Stinson. *Cryptography: Theory and Practice.* CRC Press, 1995.

CHAPTER 26

Secure Multi-Party Computation

Chapter Goals

- To introduce the concept of multi-party computation.
- To present a two-party protocol based on Yao's garbled circuit construction.
- To present a multi-party protocol based on Shamir secret sharing.

1. Introduction

Secure multi-party computation is an area of cryptography which deals with two or more players computing a function on their private inputs. They wish to do so in a way which means that their private inputs still remain private. Of course depending on the function being computed, some information about the inputs may leak. The classical example is the so-called millionaires problem; suppose a bunch of millionaires have a lunch time meeting at an expensive restaurant and decide that the richest of them will pay the bill. However, they do not want to reveal their actual wealth to each other. This is an example of a secure multi-party computation. The inputs are the values x_i, which denote the wealth of each party, and the function to be computed is

$$f(x_1, \ldots, x_n) = i \text{ where } x_i > x_j \text{ for all } i \neq j.$$

Clearly if we compute such a function then some information about party i's value leaks, i.e. that it is greater than all the other values. However, we require in secure multi-party computation that this is the only information which leaks.

One can consider a number of our previous protocols as being examples of secure multi-party computation. For example, the voting protocol given previously involves the computation of the result of each party voting, without anyone learning the vote being cast by a particular party.

One solution to securely evaluating a function is for all the parties to send their inputs to a trusted third party. This trusted party then computes the function and passes the output back to the parties. However, we want to remove such a trusted third party entirely. Intuitively a multi-party computation is said to be secure if the information which is leaked is precisely that which would have leaked if the computation had been conducted by encrypting messages to a trusted third party.

This is not the only security issue which needs to be addressed when considering secure multi-party computation. There are two basic security models: In the first model the parties are guaranteed to follow the protocols, but are interested in breaking the privacy of their fellow participants. Such adversaries are called honest-but-curious, and they in some sense correspond to passive adversaries in other areas of cryptography. Whilst honest-but-curious adversaries follow the protocol, a number of them could combine their different internal data so as to subvert the security of the non-corrupt parties. In the second model the adversaries can deviate from the protocol and may wish to pass incorrect data around so as to subvert the computation of the function. Again we allow such adversaries to talk to each other in a coalition. Such adversaries are called malicious.

There is a problem though, if we assume that communication is asynchronous, which is the most practically relevant situation, then some party must go last. In such a situation one party may have learnt the outcome of the computation, but one party may not have the value yet (namely the party which receives the last message). Any malicious party can clearly subvert the protocol by not sending the last message. Usually malicious adversaries are assumed not to perform such an attack. A protocol which is said to be secure against an adversary which can delete the final message is said to be fair.

In what follows we shall mainly explain the basic ideas behind secure multi-party computation in the case of honest-but-curious adversaries. We shall touch on the case of malicious adversaries for one of our examples though, as it provides a nice example of an application of various properties of Shamir secret sharing.

If we let n denote the number of parties which engage in the protocol, we would like to create protocols for secure multi-party computation which are able to tolerate a large number of corrupt parties. It turns out that there is a theoretical limit as to the number of parties which can be tolerated as being corrupt. For the case of honest-but-curious adversaries we can tolerate up to $n/2$ corrupt parties. However, for malicious adversaries we can tolerate up to $n/2$ corrupt parties only if we base our security on some computational assumption, for example the inability to break a symmetric encryption scheme. If we are interested in perfect security then we can only tolerate up to $n/3$ corrupt parties in the malicious case.

Protocols for secure multi-party computation usually fall into one of two distinct families. The first is based on an idea of Yao called a garbled circuit or Yao circuit, in this case one presents the function to be computed as a binary circuit, and then one "encrypts" the gates of this circuit to form the garbled circuit. This approach is clearly based on a computational assumption, i.e. that the encryption scheme is secure. The second approach is based on secret sharing schemes; here one usually represents the function to be computed as an arithmetic circuit. In this second approach one uses a perfect secret sharing scheme to obtain perfect security.

It turns out that the first approach seems better suited to the case where there are two parties, whilst the second approach is better suited to the case of three of more parties. In our discussion below we will present a computationally secure solution for the two party case in the presence of honest-but-curious adversaries, based on Yao circuits. This approach can be extended to more than two parties, and to malicious adversaries, but doing this is beyond the scope of this book. We then present a protocol for the multi-party case which is perfectly secure. We sketch two versions, one which provides security against honest-but-curious adversaries and one which provides security against malicious adversaries.

2. The Two-Party Case

We shall in this section consider the method of secure multi-party computation based on garbled circuits. We suppose there are two parties A and B each of whome have input x and y, and that A wishes to compute $f_A(x, y)$ and B wishes to compute $f_B(x, y)$. Recall this needs to be done without B learning anything about x or $f_A(x, y)$, except from what he can deduce from $f_B(x, y)$ and y, with a similar privacy statement applying to A's input and outputs.

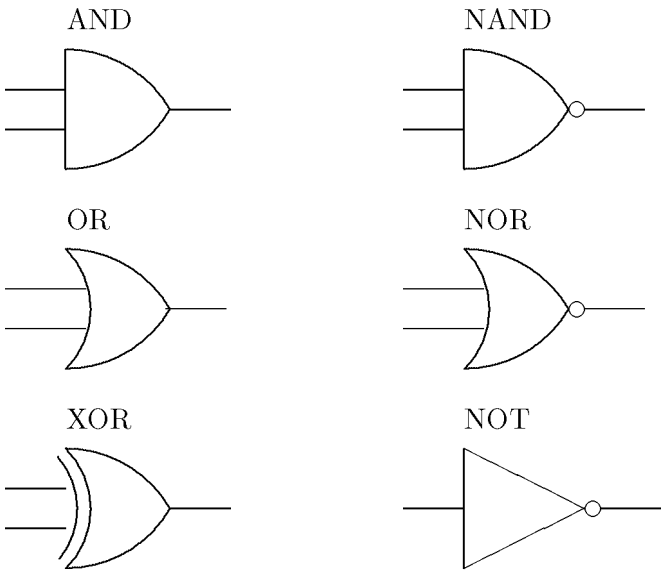
First note that it is enough for B to receive the output of a related function f. To see this we let A have an extra secret input k which is as long as the maximum output of her function $f_A(x, y)$. If we can create a protocol in which B learns the value of the function

$$f(x, y, k) = (k \oplus f_A(x, y), f_B(x, y)),$$

then B simply sends the value of $k \oplus f_A(x, y)$ back to A who can then decrypt it using k, and so determine $f_A(x, y)$. Hence, we will assume that there is only one function which needs to be computed and that its output will be determined by B.

So suppose $f(x, y)$ is the function which is to be computed, we will assume that $f(x, y)$ is a function which can be computed in polynomial time. There is therefore also a polynomial sized binary circuit which will also compute the output of the function. In the forthcoming example we will be writing out such a circuit, and so in Figure 1, we recall the standard symbols for a binary circuit.

FIGURE 1. The Basic Logic Gates



A binary circuit can be represented by a collection of wires $W = \{w_1, \ldots, w_n\}$ and a collection of gates $G = \{g_1, \ldots, g_m\}$. Each gate is function which takes as input the values of two wires, and produces the value of the output wire. For example suppose g_1 is an AND gate which takes as input wire w_1 and w_2 and produces the output wire w_3. Then gate g_1 can be represented by the following truth table.

w_1	w_2	w_3
0	0	0
0	1	0
1	0	0
1	1	1

In other words the gate g_i represents a function such that

$$0 = g_i(0, 0) = g_i(1, 0) = g_i(0, 1) \text{ and } 1 = g_i(1, 1).$$

In Yao's garbled circuit construction for secure multi-party computation a circuit is encrypted as follows:

- For each wire w_i two random cryptographic keys are selected k_i^0 and k_i^1. The first one represents the encryption of the zero value, and the second represents the encryption of the one value.
- For each wire a random value $\rho_i \in \{0, 1\}$ is chosen. This is used to also encrypt the actual wire value. If the actual wire value is v_i then the encrypted, or "external" value, is given by $\mathbb{C}e_i = v_i \oplus \rho_i$.
- For each gate we compute a "garbled table" representing the function of the gate on these encrypted values. Suppose g_i is a gate with input wires w_{i_0} and w_{i_1} and output wire w_{i_2},

then the garbled table is the following four values, for some encryption function E,

$$c_{a,b}^{w_{i_2}} = E_{k_{w_{i_0}}^{a\oplus\rho_{i_0}}, k_{w_{i_1}}^{b\oplus\rho_{i_1}}}\left(k_{w_{i_2}}^{o_{a,b}} \| o_{a,b} \oplus \rho_{i_2}\right) \text{ for } a, b \in \{0,1\}.$$

where $o_{a,b} = g_i(a \oplus \rho_{i_0}, b \oplus \rho_{i_1})$.

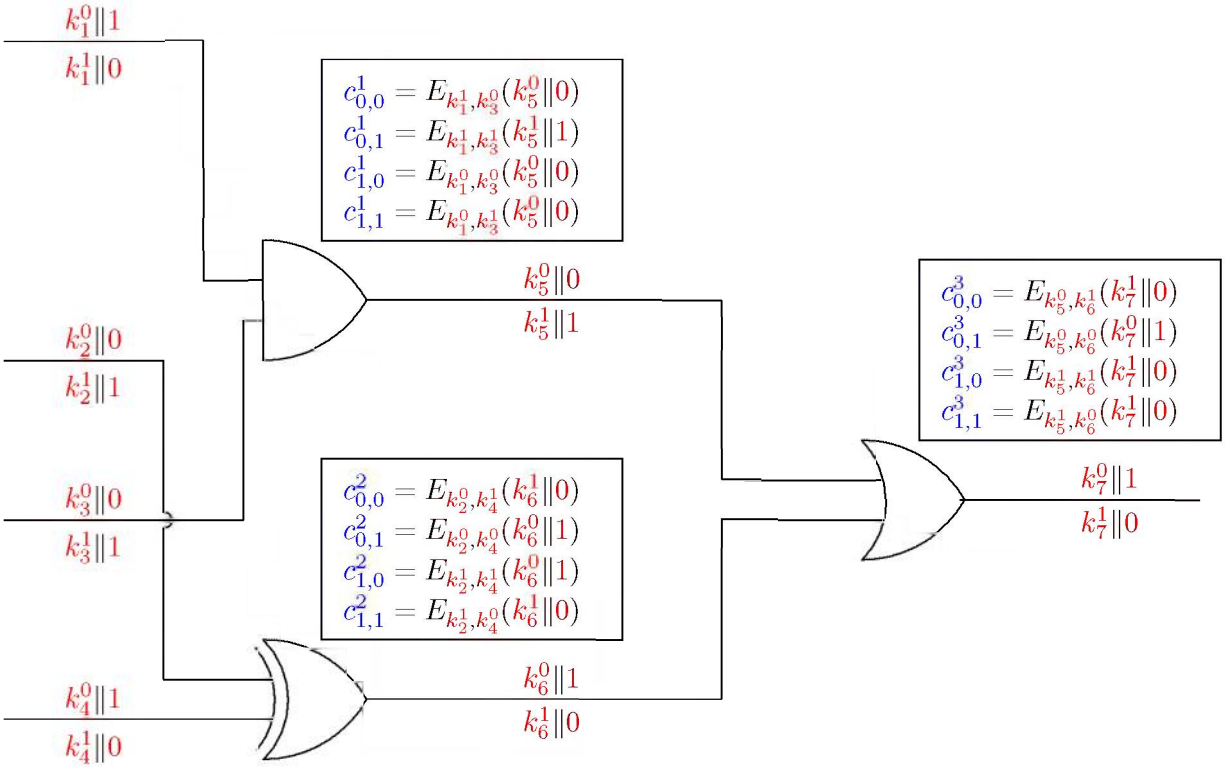
We do not consider exactly what encryption function is chosen, such a discussion is slightly beyond the scope of this book. If you want further details then look in the references at the end of this chapter, or just assume we take an encryption scheme which is suitably secure.

The above may seem rather confusing so we illustrate the method for constructing the garbled circuit with an example. Suppose A and B each have as input two bits, we shall denote A input wires by w_1 and w_2, whilst B's input wires we shall denote by w_3 and w_4. Suppose they now wish to engage in a secure multi-party computation so that B learns the value of the function

$$f(\{w_1, w_2\}, \{w_3, w_4\}) = (w_1 \wedge w_3) \vee (w_2 \oplus w_4).$$

A circuit to represent this function is given in Figure 2.

FIGURE 2. The Garbled Circuit



In Figure 2 we present the garbled values of each wire and the corresponding garbled tables representing each gate. In this example we have the following values of ρ_i,

$$\rho_1 = \rho_4 = \rho_6 = \rho_7 = 1 \text{ and } \rho_2 = \rho_3 = \rho_5 = 0.$$

Consider the first wire, the two garbled values of the wire are k_1^0 and k_1^1, which represent the 0 and 1 value respectively. Since $\rho_1 = 1$ then the external value of the internal 0 value is 1 and the external value of the internal 1 value is 0. Thus we represent the value garbled value of the wire by the pair of pairs

$$(k_1^0\|1, k_1^1\|0).$$

Now we look at the gates, and in particular consider the first gate. The first gate is an AND gate which takes as input the first and third wires. The first entry in this table corresponds to $a = b = 0$. Now the ρ values for the first and third wires are 1 and 0 respectively. Hence, the first entry in the table corresponds to what should happen if the keys k_1^1 and k_3^0 are seen; since

$$1 = 1 \oplus 0 = \rho_1 \oplus a \text{ and } 0 = 0 \oplus 0 = \rho_3 \oplus b.$$

Now the AND gate should produce the 0 output on input of 1 and 0, thus the thing which is encrypted in the first line is the key representing the zero value of the fifth wire, i.e. k_5^0, plus the "external value" of 0, namely $0 = 0 \oplus 0 = 0 \oplus \rho_5$.

We are now in a position to describe Yao's protocol. The protocol proceeds in five phases as follows:

(1) Party A generates the garbled circuit as above, and transmits to party B only the values $c_{a,b}^i$.
(2) Party A then transmits to party B the garbled values of the component of its input wires. For example, suppose in our example that party A's input is $w_1 = 0$ and $w_2 = 0$. Then party A transmits to party B the two values
$$k_1^0 \| 1 \text{ and } k_2^0 \| 0.$$
Note that party B cannot learn the actual values of w_1 and w_2 from these values since it does not know ρ_1 and ρ_2, and the keys k_1^0 and k_2^0 just look like random keys.
(3) Party A and B then engage in an oblivious transfer protocol as in Section 3, for each of party B's input wires. In our example suppose that party B's input is $w_3 = 1$ and $w_4 = 0$. The two parties execute two oblivious transfer protocols, one with A's input
$$k_3^0 \| 0 \text{ and } k_3^1 \| 1,$$
and B's input 1, and one with A's input
$$k_4^0 \| 1 \text{ and } k_4^1 \| 0,$$
and B's input 0. At the end of this oblivious transfer phase party B has learnt
$$k_3^1 \| 1 \text{ and } k_4^0 \| 1.$$
(4) Party A then transmits to party B the values of ρ_i for all of the output wires. In our example he reveals the value of $\rho_7 = 1$.
(5) Finally party B evaluates the circuit using the garbled input wire values he has been given.

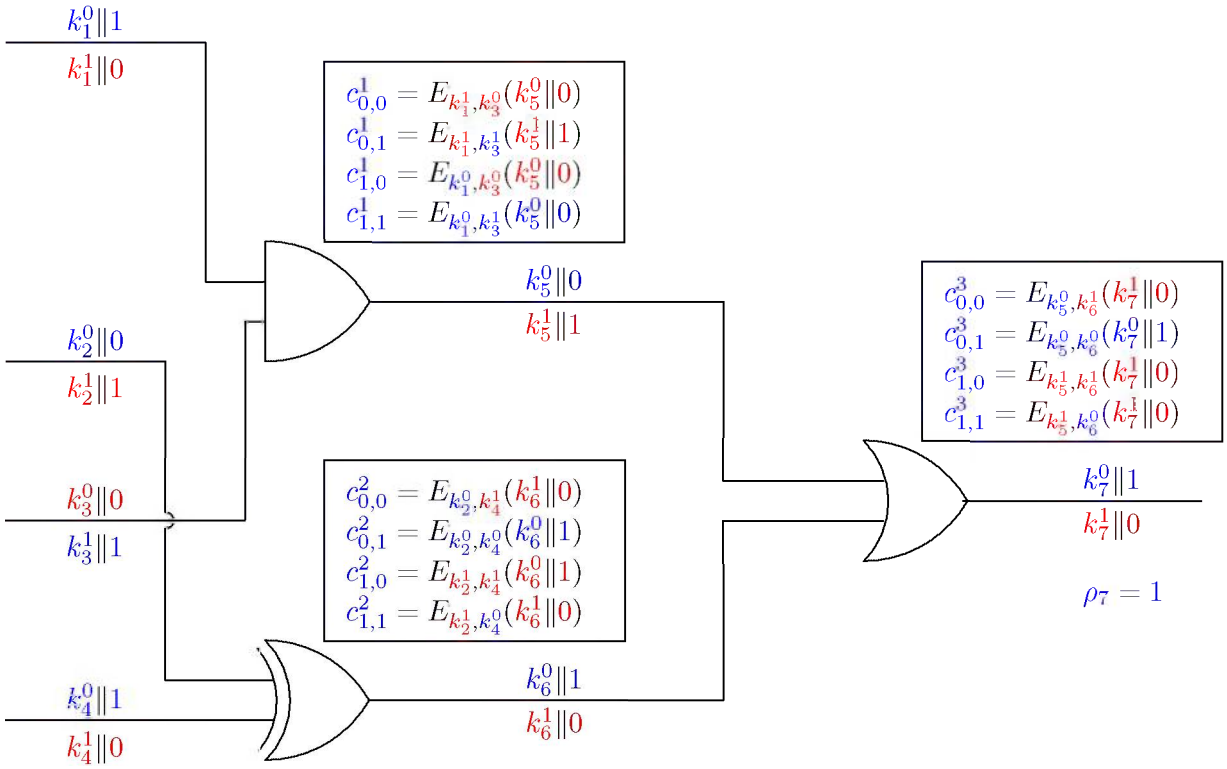
In summary, in the first stage party B only knows the garbled circuit as in the blue items in Figure 2, but by the last stage he knows the blue items in Figure 3.

We now describe how the circuit is evaluated in detail. Please refer to Figure 3 for a graphical description of this. Firstly party B evaluates the AND gate, he knows that the external value of wire one is 1 and the external value of wire three is 1. Thus he looks up the entry $c_{1,1}^1$ in the table and decrypts it using the two keys he knows, i.e. k_1^0 and k_3^1. He then obtains the values $k_5^0 \| 0$. He has no idea whether this represents the zero or one value of the fifth wire, since he has no idea as to the value of ρ_5.

Party B then performs the same operation with the XOR gate. This has input wire 2 and wire 4, for which party B knows that the external values are 0 and 1 respectively. Thus party B decrypts the entry $c_{0,1}^2$ to obtain $k_6^0 \| 1$.

A similar procedure is then carried out with the final OR gate, using the keys and external values of the fifth and sixth wires. This results in a decryption which reveals the value $k_7^0 \| 1$. So the external value of the seventh wire is equal to 1, but party B has been told that $\rho_7 = 1$, and hence the internal value of wire seven will be $0 = 1 \oplus 1$. Hence, the output of the function is the bit 0.

FIGURE 3. Evaluating The Garbled Circuit



So what has Party B learnt from the secure multi-party computation? Party B knows that the output of the final OR gate is zero, which means that the inputs must also be zero, which means that the output of the AND gate is zero and the output of the XOR gate is zero. However, party B knows that the output of the AND gate will be zero, since its own input was zero. However, party B has learnt that party A's second input wire represented zero, since otherwise the XOR gate would not have output zero. So whilst party A's first input remains private, the second input does not. This is what we meant by a protocol keeping the inputs private, bar what could be deduced from the output of the function.

3. The Multi-Party Case: Honest-but-Curious Adversaries

The multi-party case is based on using a secret sharing scheme to evaluate an arithmetic circuit. An arithmetic circuit consists of a finite field $\mathbb{F}_q$, and a polynomial function (which could have many inputs and outputs) defined over the finite field. The idea being that such a function can be evaluated by executing a number of addition and multiplication gates over the finite field.

Given an arithmetic circuit it is clear one could express it as a binary circuit, by simply expanding the addition and multiplication gates of the arithmetic circuit out as their binary circuit equivalents. One can also represent every binary circuit as an arithmetic circuit, since every gate in the binary circuit can be represented as a linear function of the input values to the gate, and their products. Whilst the two representations are equivalent it is clear that some functions are easier to represent as binary circuits and some are easy to represent as arithmetic circuits.

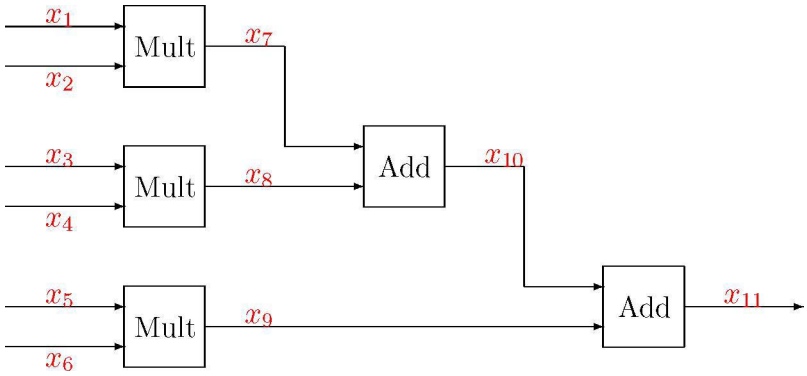
As before we shall present the protocol via a running example. We shall suppose we have six parties, $P_1, \ldots, P_6$ who have six secret values $x_1, \ldots, x_6$ each of which lie in $\mathbb{F}_p$, for some reasonably large prime p. For example we could take $p \approx 2^{128}$, but in our example to make things easier

to represent we will take $p = 101$. The parties are assumed to want to compute the value of the function

$$f(x_1, \ldots, x_6) = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6 \pmod{p}.$$

Hence, the arithmetic circuit for this function consists of three multiplication gates and two addition gates, as in Figure 4, where we label the intermediate values as numbered "wires".

FIGURE 4. Graphical Representation of the Example Arithmetic Circuit



The basic protocol idea is as follows, where we use Shamir secret sharing as our secret sharing scheme. The value of each wire x_i is shared between all players, with each player j obtaining a share $x_i{}^{(j)}$. Clearly, if enough players come together then they can determine the value of the wire x_i, by the properties of the secret sharing scheme, and each player can deal shares of his or her own values at the start of the protocol.

The main problem is how to obtain the shares of the outputs of the gates, given shares of the inputs of the gates. Recall in Shamir secret sharing the shared value is given by the constant term of a polynomial f of degree t, with the sharings being the evaluation of the polynomial at given positions corresponding to each participant $f(i)$.

First we consider how to compute the `Add` gates. Suppose we have two secrets a and b which are shared using the polynomials

$$f(X) = a + f_1 X + \cdots + f_t X^t,$$
$$g(X) = b + g_1 X + \cdots + g_t X^t.$$

Each of our parties has a share $a^{(i)} = f(i)$ and $b^{(i)} = g(i)$. Now consider the polynomial

$$h(X) = f(X) + g(X).$$

This polynomial provides a sharing of the sum $c = a + b$, and we have

$$c^{(i)} = h(i) = f(i) + g(i) = a^{(i)} + b^{(i)}.$$

Hence, the parties can compute a sharing of the output of an `Add` gate without any form of communication between them.

Computing the output of a `Mult` gate is more complicated. First we recap on the following property of Lagrange interpolation. If $f(X)$ is a polynomial and we distribute the values $f(j)$ then there is a vector, called the recombination vector, $(r_1, \ldots, r_n)$ such that

$$f(0) = \sum_{i=1}^{n} r_i \cdot f(i).$$

And the same vector works for all polynomials $f(X)$ of degree at most $n-1$.

To compute the `Mult` gate we perform the following four steps. We assume as input that each party has a share of a and b via $a^{(i)} = f(i)$ and $b^{(i)} = g(i)$, where $f(0) = a$ and $g(0) = b$. We wish to compute a sharing $c^{(i)} = h(i)$ such that $h(0) = c = a \cdot b$.

- Each party locally computes $d^{(i)} = a^{(i)} \cdot b^{(i)}$.
- Each party produces a polynomial $\delta_i(X)$ of degree at most t such that $\delta_i(0) = d^{(i)}$.
- Each party i distributes to party j the value $d_{i,j} = \delta_i(j)$.
- Each party j computes $c^{(j)} = \sum_{i=1}^{n} r_i \cdot d_{i,j}$.

So why does this work? Consider the first step, here we are actually effectively computing a polynomial $h'(X)$ of degree at most $2 \cdot t$, with $d^{(i)} = h'(i)$, and $c = h'(0)$. Hence, the only problem with the sharing in the first step is that the underlying polynomial has too high a degree. The main thing to note, is that if

$$2 \cdot t \leq n - 1 \tag{16}$$

then we have $c = \sum_{i=1}^{n} r_i \cdot d^{(i)}$.

Now consider the polynomials $\delta_i(X)$ generated in the second step, and consider what happens when we recombine them using the recombination vector, i.e. set

$$h(X) = \sum_{i=1}^{n} r_i \cdot \delta_i(X).$$

Since the $\delta_i(X)$ are all of degree at most t, the polynomial $h(X)$ is also of degree at most t. We also have that

$$h(0) = \sum_{i=1}^{n} r_i \cdot \delta_i(0) = \sum_{i=1}^{n} r_i \cdot d^{(i)} = c,$$

assuming $2 \cdot t \leq n - 1$. Thus $h(X)$ is a polynomial which *could* be used to share the value of the product. Not only that, but it *is* the polynomial underlying the sharing produced in the final step. To see this notice that

$$h(j) = \sum_{i=1}^{n} r_i \cdot \delta_i(j) = \sum_{i=1}^{n} r_i \cdot d_{i,j} = c^{(j)}.$$

So assuming $t < n/2$ we can produce a protocol which evaluates the arithmetic circuit correctly. We illustrate the method by examining what would happen for our example circuit in Figure 4, with $p = 101$. Recall there are six parties; we shall assume that their inputs are given by

$$x_1 = 20,\ x_2 = 40,\ x_3 = 21,\ x_4 = 31,\ x_5 = 1,\ x_6 = 71.$$

Each party first computes a sharing $x_i^{(j)}$ of their secret amongst the six parties. They do this by each choosing a random polynomial of degree $t = 2$ and evaluating it at $j = 1, 2, 4, 5, 6$. The values obtained are then distributed, securely to each party. Hence, each party obtains its row of the following table

	i					
j	1	2	3	4	5	6
1	44	2	96	23	86	83
2	26	0	63	13	52	79
3	4	22	75	62	84	40
4	93	48	98	41	79	10
5	28	35	22	90	37	65
6	64	58	53	49	46	44

As an exercise you should work out the associated polynomials corresponding to each column.

The parties then engage in the multiplication protocol so as to compute sharings of $x_7 = x_1 \cdot x_2$. They first compute their local multiplication, by each multiplying the first two elements in their row of the above table, then they for a sharing of this local multiplication. These sharings of six numbers between six parties are then distributed securely as before. In our example run each party, for this multiplication, obtains the sharings given by the column of the following table

	j					
i	1	2	3	4	5	6
1	92	54	20	91	65	43
2	10	46	7	95	7	46
3	64	100	96	52	69	46
4	23	38	41	32	11	79
5	47	97	77	88	29	1
6	95	34	11	26	79	69

Each party then takes the six values obtained and recovers their share of the value of x_7. We find that the six shares of x_7 are given by

$$x_7{}^{(1)} = 9,\ x_7{}^{(2)} = 97,\ x_7{}^{(3)} = 54,\ x_7{}^{(4)} = 82,\ x_7{}^{(5)} = 80,\ x_7{}^{(6)} = 48.$$

Repeating the multiplication protocol twice more we also obtain a sharing of x_8 as

$$x_8{}^{(1)} = 26,\ x_8{}^{(2)} = 91,\ x_8{}^{(3)} = 38,\ x_8{}^{(4)} = 69,\ x_8{}^{(5)} = 83,\ x_8{}^{(6)} = 80,$$

and x_9 as

$$x_9{}^{(1)} = 57,\ x_9{}^{(2)} = 77,\ x_9{}^{(3)} = 30,\ x_9{}^{(4)} = 17,\ x_9{}^{(5)} = 38,\ x_9{}^{(6)} = 93,$$

We are then left with the two addition gates to produce the sharings of the wires x_{10} and x_{11}. These are obtained by locally adding together the various shared values so that

$$x_{11}{}^{(1)} = x_7{}^{(1)} + x_8{}^{(1)} + x_9{}^{(1)} \pmod{101} = 9 + 26 + 57 \pmod{101} = 92,$$

etc, to obtain

$$x_{11}{}^{(1)} = 92,\ x_{11}{}^{(2)} = 63,\ x_{11}{}^{(3)} = 21,\ x_{11}{}^{(4)} = 67,\ x_{11}{}^{(5)} = 100,\ x_{11}{}^{(6)} = 19.$$

The parties then make public these shares, and recover the hidden polynomial, of degree $t = 2$, which produces these sharings, namely

$$7 + 41X^2 + 44X^2.$$

Hence, the result of the multi-party computation is the value 7.

Now assume that more than t parties are corrupt, in the sense that they collude to try and break the privacy of the non-corrupted parties. The corrupt parties can now come together and recover any of the underlying secrets in the scheme, since we have used Shamir secret sharing using polynomials of degree at most t. It can be shown that, using the perfect secrecy of the Shamir secret sharing scheme that as long as less than or equal to t parties are corrupt then the above protocol is perfectly securely.

However, it is only perfectly secure assuming all parties follow the protocol, i.e. we are in the honest-but-curious model. As soon as we allow parties to deviate from the protocol then they can force the honest parties to produce invalid results. To see this just notice that a dishonest party could simply produce an invalid sharing of its product in the second part of the multiplication protocol above.

4. The Multi-Party Case: Malicious Adversaries

To produce a scheme which is secure against such active adversaries we need to force all parties to either follow the protocol, or we should be able to recover from errors which malicious parties introduce introduce into the protocol. It is the second of these two approaches which we shall follow in this section, by using the error correction properties of the Shamir secret sharing scheme.

As already remarked the above protocol is not secure against malicious adversaries, due to the ability of an attacker to make the multiplication protocol output an invalid answer. To make the above protocol secure against malicious adversaries we make use of various properties of the Shamir secret sharing scheme.

The protocol runs in two stages: The preprocessing stage does not involve any of the secret inputs of the parties, it purely depends on the number of multiplication gates in the circuit. In the main phase of the protocol the circuit is evaluated as in the previous section, but using a slightly different multiplication protocol. Malicious parties can force the preprocessing stage to fail, however if it completes then the honest parties will be able to evaluate the circuit as required.

The preprocessing phase runs as follows. First using the techniques from Chapter 23 a pseudorandom secret sharing scheme, PRSS, and a pseudorandom zero sharing scheme, PRZS, are set up. Then for each multiplication gate in the circuit we compute a random triple of sharings $a^{(i)}$, $b^{(i)}$ and $c^{(i)}$ such that $c = a \cdot b$. This is done as follows:

- Using PRSS generate two random sharings, $a^{(i)}$ and $b^{(i)}$, of degree t.
- Using PRSS generate another random sharing $r^{(i)}$ of degree t.
- Using PRZS generate a sharing $z^{(i)}$, of degree $2 \cdot t$ of zero.
- Each player then locally computes
$$s^{(i)} = a^{(i)} \cdot b^{(i)} - r^{(i)} + z^{(i)}.$$
Note this local computation will produce a degree $2 \cdot t$ sharing of the value $s = a \cdot b - r$.
- Then the players broadcast their values $s^{(i)}$ and try to recover s. This can be done always using the error correction properties of Reed–Solomon codes assuming the number of malicious parties is bounded by $n/3$.
- Now the players locally compute the shares $c^{(i)}$ from $c^{(i)} = s + r^{(i)}$.

Assuming the above preprocessing phase completes successfully all we need do is specify how the parties implement a `Mult` in the presence of malicious adversaries. We assume the inputs to the multiplication gate are given by $x^{(i)}$ and $y^{(i)}$ and we wish to compute a sharing $z^{(i)}$ of the produce $z = x \cdot y$. From the preprocessing stage, the parties also have for each gate, a triple of shares $a^{(i)}$, $b^{(i)}$ and $c^{(i)}$ such that $c = a \cdot b$. The protocol for the multiplication is then as follows:

- Compute locally, and then broadcast, the values $d^{(i)} = x^{(i)} - a^{(i)}$ and $e^{(i)} = y^{(i)} - b^{(i)}$
- Reconstruct the values of $d = x - a$ and $d = y - b$.
- Locally compute the shares
$$z^{(i)} = d \cdot e + d \cdot b^{(i)} + e \cdot a^{(i)} + c^{(i)}.$$

Note that the reconstruction in the second step can be completed as long as there are at most $t < n/3$ malicious parties. The computation in the last step recovers the valid shares due to the linear nature of the Shamir secret sharing scheme and the underlying equation

$$\begin{aligned} d \cdot e + d \cdot b + e \cdot a + c &= (x - a) \cdot (y - b) + (x - a) \cdot b + (y - b) \cdot a + c \\ &= ((x - a) + a) \cdot ((y - b) + b) \\ &= x \cdot y = z. \end{aligned}$$

Chapter Summary

- We have explained how to perform two party secure computation in the precense of honest-but-curious adversaries using Yao's garbled circuit construction.
- For the many party case we have presented a protocol based on evaluating arithmetic, as opposed to binary, circuits which is based on Shamir secret sharing.
- The main issue with this latter protocol is how to evaluate the multiplication gates. We presented two methods: The first, simpler, method is applicable when one is only dealing with honest-but-curious adversaries, the second, more involved, method is for the case of malicious adversaries.

Further Reading

The original presentation of Yao's idea appears in FOCS 1986, It can be transformed into a scheme for malicious adversaries using a general technique of Goldreich et. al. The discussion of the secret sharing based solution for the honest and malicious cases closely follows the treatment in Damgård et. al.

I. Damgård, M. Geisler, M. Kroigaard and J.B. Nielsen. *Asynchronous multiparty computation: Theory and implementation* IACR e-print `eprint.iacr.org/2008/415`.

O. Goldreich, S. Micali and A. Wigderson. *How to play any mental game.* In Symposium on Theory of Computing – STOC 1987, ACM, 218–229, 1987.

A.C. Yao. *How to generate and exchange secrets.* In Foundations of Computer Science – FOCS 1986, IEEE, 162–167, 1986.

APPENDIX A

Basic Mathematical Terminology

This appendix is presented as a series of notes which summarizes most of the mathematical terminology needed in this book. In this appendix we present the material in a more formal manner than we did in Chapter 1 and the rest of the book.

1. Sets

Here we recap on some basic definitions etc. which we list here for completeness.

DEFINITION A.1. *For two sets A, B we define the union, intersection, difference and cartesian product by*

$$A \cup B = \{x : x \in A \text{ or } x \in B\},$$
$$A \cap B = \{x : x \in A \text{ and } x \in B\},$$
$$A \setminus B = \{x : x \in A \text{ and } x \notin B\},$$
$$A \times B = \{(x, y) : x \in A \text{ and } y \in B\}.$$

The statement $A \subset B$ means that

$$x \in A \Rightarrow x \in B.$$

Using these definitions one can prove in a standard way all the basic results of set theory that one shows in school using Venn diagrams. For example

LEMMA A.2. *If $A \subset B$ and $B \subset C$ then $A \subset C$.*

PROOF. Let x be an element of A, we wish to show that x is an element of C. Now as $A \subset B$ we have that $x \in B$, and as $B \subset C$ we then deduce that $x \in C$. □

Notice that this is a proof whereas an argument using Venn diagrams is not a proof. Using Venn diagrams merely shows you were not clever enough to come up with a picture which proved the result false.

2. Relations

Next we define relations and some properties that they have. Relations, especially equivalence relations, play an important part in algebra and it is worth considering them at this stage so it is easier to understand what is going on later.

DEFINITION A.3. *A (binary) relation on a set A is a subset of the Cartesian product $A \times A$.*

This we explain with an example:
Consider the relationship 'less than or equal to' between natural numbers. This obviously gives us the set

$$LE = \{(x, y) : x, y \in \mathbb{N},\ x \text{ is less than or equal to } y\}.$$

In much the same way every relationship that you have met before can be written in this set-theoretic way. An even better way to put the above is to define the relation less than or equal to to be the set

$$LE = \{(x, y) : x, y \in \mathbb{N},\ x - y \notin \mathbb{N} \setminus \{0\}\}.$$

Obviously this is a very cumbersome notation so for a relation R on a set S we write

$$x \; R \; y$$

if $(x, y) \in R$, i.e. if we now write $\leq$ for LE we obtain the usual notation $1 \leq 2$ etc.

Relations which are of interest in mathematics usually satisfy one or more of the following four properties:

DEFINITION A.4.
A relation R on a set S is reflexive if for all $x \in S$ we have $(x, x) \in R$.
A relation R on a set S is symmetric if $(x, y) \in R$ implies that $(y, x) \in R$.
A relation R on a set S is anti-symmetric if $(x, y) \in R$ and $(y, x) \in R$ implies that $x = y$.
A relation R on a set S is transitive if $(x, y) \in R$ and $(y, z) \in R$ implies that $(x, z) \in R$.

We return to our example of $\leq$. This relation $\leq$ is certainly reflexive as $x \leq x$ for all $x \in \mathbb{N}$. It is not symmetric as $x \leq y$ does not imply that $y \leq x$, however it is anti-symmetric as $x \leq y$ and $y \leq x$ imply that $x = y$. You should note that it is transitive as well.

Relations like $\leq$ occur so frequently that we give them a name:

DEFINITION A.5. *A relation which is reflexive, transitive and anti-symmetric is called a partial order relation.*

DEFINITION A.6. *A relation which is transitive and anti-symmetric and which for all x, y we have either $(x, y) \in R$ or $(y, x) \in R$ is called a total order relation.*

Another important type of relationship is that of an equivalence relation:

DEFINITION A.7. *A relation which is reflexive, symmetric and transitive is called an equivalence relation.*

The obvious example of $\mathbb{N}$ and the relation 'is equal to' is an equivalence relation and hence gives this type of relation its name. One of the major problems in any science is that of classification of sets of objects. This amounts to placing the objects into mutually disjoint subsets. An equivalence relation allows us to place equivalent elements into disjoint subsets. Each of these subsets is called an equivalence class. If the properties we are interested in are constant over each equivalence class then we may as well restrict our attention to the equivalence classes themselves. This often leads to greater understanding. In the jargon this process is called factoring out by the equivalence relation. It occurs frequently in algebra to define new objects from old, e.g. quotient groups. The following example is probably the most familiar being a description of modular arithmetic:

Let m be a fixed positive integer. Consider the equivalence relation on $\mathbb{Z}$ which says x is related to y if $(x - y)$ is divisible by m. This is an equivalence relation, which you should check. The equivalence classes we denote by

$$\begin{aligned}
\overline{0} &= \{\ldots, -2m, -m, 0, m, 2m, \ldots\}, \\
\overline{1} &= \{\ldots, -2m+1, -m+1, 1, m+1, 2m+1, \ldots\}, \\
\ldots \quad &\quad \ldots \\
\overline{m-1} &= \{\ldots, -m-1, -1, m-1, 2m-1, 3m-1, \ldots\}.
\end{aligned}$$

Note that there are m distinct equivalence classes, one for each of the possible remainders on division by m. The classes are often called the residue classes modulo m. The resulting set $\{\overline{0}, \ldots, \overline{m-1}\}$ is often denoted by $\mathbb{Z}/m\mathbb{Z}$ as we have divided out by all multiples of m. If m is a prime number, say p, then the resulting set is often denoted $\mathbb{F}_p$ as the resulting object is a field.

3. Functions

We give two definitions of functions; the first is wordy and is easier to get hold of, the second is set-theoretic.

DEFINITION A.8. *A function is a rule which maps the elements of one set, the domain, with those of another, the codomain. Each element in the domain must map to one and only one element in the codomain.*

The point here is that the function is not just the rule, e.g. $f(x) = x^2$, but also the two sets that one is using. A few examples will suffice.

(1) The rule $f(x) = \sqrt{x}$ is not a function from $\mathbb{R}$ to $\mathbb{R}$ since the square root of a negative number is not in $\mathbb{R}$. It is also not a function from $\mathbb{R}_{\geq 0}$ to $\mathbb{R}$ since every element of the domain has two square roots in the codomain. But it is a function from $\mathbb{R}_{\geq 0}$ to $\mathbb{R}_{\geq 0}$.
(2) The rule $f(x) = 1/x$ is not a function from $\mathbb{R}$ to $\mathbb{R}$ but it is a function from $\mathbb{R} \setminus \{0\}$ to $\mathbb{R}$.
(3) Note not every element of the codomain need have an element mapping to it. Hence, the rule $f(x) = x^2$ taking elements of $\mathbb{R}$ to elements of $\mathbb{R}$ is a function.

Our definition of a function is unsatisfactory as it would also require a definition of what a rule is. In keeping with the spirit of everything else we have done we give a set-theoretic description.

DEFINITION A.9. *A function from the set A to the set B is a subset F of $A \times B$ such that:*

(1) *If $(x, y) \in F$ and $(x, z) \in F$ then $y = z$.*
(2) *For all $x \in A$ there exists a $y \in B$ such that $(x, y) \in F$.*

The set A is called the domain, the set B the codomain. The first condition means that each element in the domain maps to at most one element in the codomain. The second condition means that each element of the domain maps to at least one element in the codomain. Given a function f from A to B and an element x of A then we denote by $f(x)$ the unique element in B such that $(x, f(x)) \in f$.

One can compose functions, if the definitions make sense. Say one has a function f from A to B and a function g from B to C then the function $g \circ f$ is the function with domain A and codomain C consisting of the elements $(x, g(f(x)))$.

LEMMA A.10. *Let f be a function from A to B, let g be a function from B to C and h be a function from C to D, then we have*

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

PROOF. Let (a, d) belong to $(h \circ g) \circ f$. Then their exists an $(a, b) \in f$ and a $(b, d) \in (h \circ g)$ for some $b \in B$, by definition of composition of functions. Again by definition there exists a $c \in C$ such that $(b, c) \in g$ and $(c, d) \in h$. Hence $(a, c) \in (g \circ f)$, which shows $(a, d) \in h \circ (g \circ f)$. Hence

$$(h \circ g) \circ f \subset h \circ (g \circ f).$$

Similarly one can show the other inclusion. □

One function, the identity function, is particularly important:

DEFINITION A.11. *The identity function* id_A *on a set A is the set $\{(x, x) : x \in A\}$.*

LEMMA A.12. *For any function f from A to B we have*

$$f \circ \mathrm{id}_A = \mathrm{id}_B \circ f = f.$$

PROOF. Let x be an element of A, then

$$(f \circ \mathrm{id}_A)(x) = f(\mathrm{id}_A(x)) = f(x) = \mathrm{id}_B(f(x)) = (\mathrm{id}_B \circ f)(x).$$

□

Two properties that we shall use all the time are the following:

DEFINITION A.13.
A function f from A to B is said to be injective (or 1:1) if for any two elements, x, y of A with $f(x) = f(y)$ we have $x = y$.
A function f from A to B is said to be surjective (or onto) if for every element $b \in B$ there exists an element $a \in A$ such that $f(a) = b$.

A function which is both injective and surjective is called bijective (or a 1:1 correspondence). We shall now give some examples.

(1) The function from $\mathbb{R}$ to $\mathbb{R}$ given by $f(x) = x + 2$ is bijective.
(2) The function from $\mathbb{N}$ to $\mathbb{N}$ given by $f(x) = x + 2$ is injective but not surjective as the elements $\{0, 1\}$ are not the image of anything.
(3) The function from $\mathbb{R}$ to $\mathbb{R}_{\geq 0}$ given by $f(x) = x^2$ is surjective as every non-negative real number has a square root in $\mathbb{R}$ but it is not injective as if $x^2 = y^2$ then we could have $x = -y$.

The following gives us a good reason to study bijective functions.

LEMMA A.14. *A function $f : A \to B$ is bijective if and only if there exists a function $g : B \to A$ such that $f \circ g$ and $g \circ f$ are the identity function.*

We leave the proof of this lemma as an exercise. Note, applying this lemma to the resulting g means that g is also bijective. Such a function as g in the above lemma is called an inverse of f and is usually denoted f^{-1}. Note that a function only has an inverse if it is bijective.

4. Permutations

We let A be a finite set of cardinality n, without loss of generality we can assume that $A = \{1, 2, \ldots, n\}$. A bijective function from A to A is called a permutation. The set of all permutations on a set of cardinality n is denoted by S_n.

Suppose $A = \{1, 2, 3\}$, then we have the permutation $f(1) = 2$, $f(2) = 3$ and $f(3) = 1$. This is a very cumbersome way to write a permutation. Mathematicians (being lazy people) have invented the following notation, the function f above is written as

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}.$$

What should be noted about this notation (which applies for arbitrary n) is that all the numbers between 1 and n occur exactly once on each row. The first row is always given as the numbers 1 to n in increasing order. Any such matrix with these properties represents a permutation, and all permutations can be represented by such a matrix. This leads us to the elementary

LEMMA A.15. *The cardinality of the set S_n is $n!$.*

PROOF. This is a well-known argument. There are n choices for the first element in the second row of the above matrix. Then there are $n - 1$ choices for the second element in the second row and so on. □

If σ is a permutation on a set S then we usually think of σ acting on the set. So if $s \in S$ then we write

$$s^\sigma$$

for the action of σ on the element s.

Suppose we define the permutations

$$g = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$$
$$f = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

As permutations are nothing but functions we can compose them. Remembering that $g \circ f$ means apply the function f and then apply the function g we see that

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

means $1 \to 3 \to 1$, $2 \to 2 \to 3$ and $3 \to 1 \to 2$. Hence, the result of composing the above two permutations is

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}. \tag{17}$$

However, can cause confusion when using our "acting on a set" notation above. For example

$$1^{g \circ f} = g(f(1)) = 3$$

so we are unable to read the permutation from left to right. However, if we use another notation say $\cdot$ to mean

$$f \cdot g = g \circ f$$

then we are able to read the expression from left to right. We shall call this operation multiplying permutations.

Mathematicians are, as we said, by nature lazy people and this notation we have introduced is still a little too much. For instance we always write down the numbers $1, \ldots, n$ in the top row of each matrix to represent a permutation. Also some columns are redundant, for instance the first column of the permutation (17). We now introduce another notation for permutations which is concise and clear. We first need to define what a cycle is.

DEFINITION A.16. *By a cycle or n-cycle we mean the object* $(x_1, \ldots, x_n)$ *with distinct* $x_i \in \mathbb{N} \setminus \{0\}$. *This represents the permutation* $f(x_1) = x_2$, $f(x_2) = x_3$, $\ldots$, $f(x_{n-1}) = x_n$, $f(x_n) = x_1$ *and for* $x \notin \{x_1, \ldots, x_n\}$ *we have* $f(x) = x$.

For instance we have

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} = (1, 2, 3) = (2, 3, 1) = (3, 1, 2).$$

Notice that a cycle is not a unique way of representing a permutation. As another example we have

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} = (1, 3)(2) = (3, 1)(2).$$

The identity permutation is represented by $()$. Again, as mathematicians are lazy we always write $(1, 3)(2) = (1, 3)$. This can lead to ambiguities as $(1, 2)$ could represent a function from

$$\{1, 2\} \text{ to } \{1, 2\}$$

or

$$\{1, 2, \ldots, n\} \text{ to } \{1, 2, \ldots, n\}.$$

Which function it represents is usually however clear from the context.

Two cycles $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ are called disjoint if $\{x_1, \ldots, x_n\} \cap \{y_1, \ldots, y_n\} = \emptyset$. It is easy to show that if σ and τ are two disjoint cycles then

$$\sigma \cdot \tau = \tau \cdot \sigma.$$

Note this is not true for cycles which are not disjoint, e.g.

$$(1,2,3,4)\cdot(3,5) = (1,2,5,3,4) \neq (1,2,3,5,4) = (3,5)\cdot(1,2,3,4).$$

Our action of permutations on the underlying set can now be read easily from left to right,

$$2^{(1,2,3,4)\cdot(3,5)} = 3^{(3,5)} = 5 = 2^{(1,2,5,3,4)},$$

as the permutation $(1,2,3,4)$ maps 2 to 3 and the permutation $(3,5)$ maps 3 to 5.

What really makes disjoint cycles interesting is the following

LEMMA A.17. *Every permutation can be written as a product of disjoint cycles.*

PROOF. Let σ be a permutation on $\{1,\ldots,n\}$. Let σ_1 denote the cycle

$$(1,\sigma(1),\sigma(\sigma(1)),\ldots,\sigma(\ldots\sigma(1)\ldots)),$$

where we keep applying σ until we get back to 1. We then take an element x of $\{1,\ldots,n\}$ such that $\sigma_1(x) = x$ and consider the cycle σ_2 given by

$$(x,\sigma(x),\sigma(\sigma(x)),\ldots,\sigma(\ldots\sigma(x)\ldots)).$$

We then take an element of $\{1,\ldots,n\}$ which is fixed by σ_1 and σ_2 to create a cycle σ_3. We continue this way until we have used all elements of $\{1,\ldots,n\}$. The resulting cycles $\sigma_1,\ldots,\sigma_t$ are obviously disjoint and their product is equal to the cycle σ. □

What is nice about this proof is that it is constructive. Given a permutation we can follow the procedure in the proof to obtain the permutation as a product of disjoint cycles.

Consider the permutation

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 2 & 3 & 7 & 6 & 8 & 4 & 1 & 5 & 9 \end{pmatrix}.$$

We have $\sigma(1) = 2,\ \sigma(2) = 3,\ \sigma(3) = 7$ and $\sigma(7) = 1$ so the first cycle is

$$\sigma_1 = (1,2,3,7).$$

The next element of $\{1,\ldots,9\}$ which we have not yet considered is 4. We have $\sigma(4) = 6$ and $\sigma(6) = 4$ so $\sigma_2 = (4,6)$. Continuing in this way we find $\sigma_3 = (5,8)$ and $\sigma_4 = (9)$. Hence we have

$$\sigma = (1,2,3,7)(4,6)(5,8)(9) = (1,2,3,7)(4,6)(5,8).$$

5. Operations

In mathematics one meets lots of binary operations: ordinary addition and multiplication, composition of functions, matrix addition and multiplication, multiplication of permutations, etc., the list is somewhat endless. All of these binary operations have a lot in common, they also have many differences, for instance, for two real numbers x and y we have $x\cdot y = y\cdot x$, but for two 2 by 2 matrices with real entries, A and B, it is not true that we always have $A\cdot B = B\cdot A$. To study the similarities and differences between these operations we formalize the concept below. We then prove some results which are true of operations given some basic properties, these results can then be applied to any of the operations above which satisfy the given properties. Hence our abstraction will allow us to prove results in many areas at once.

DEFINITION A.18. *A (binary) operation on a set A is a function from the domain $A\times A$ to the codomain A.*

So if $A = \mathbb{R}$ we could have the function $f(x,y) = x+y$. Writing $f(x,y)$ all the time can become a pain so we often write a symbol between the x and the y to denote the operation, e.g.

$$\begin{array}{lll} x\cdot y & x+y & x\ y \\ x\circ y & x\odot y & x\diamond y \\ x\wedge y & x\vee y & x\star y. \end{array}$$

Most often we write $x + y$ and $x \cdot y$, we refer to the former as additive notation and the latter as multiplicative notation. One should bear in mind that we may not be actually referring to ordinary multiplication and addition when we use these terms/notations.

Operations can satisfy various properties:

DEFINITION A.19 (Associative). *An operation $\diamond$ is said to be associative if for all x, y and z we have*

$$(x \diamond y) \diamond z = x \diamond (y \diamond z).$$

Operations which are associative include all those examples mentioned above. Non-associative operations do exist but cryptographers have very little interest in them. Note that for an associative operation the expression

$$w \diamond x \diamond y \diamond z$$

is well defined, as long as we do not move the relative position of any of the terms it does not matter which operation we carry out first.

DEFINITION A.20 (Commutative). *An operation $\vee$ is said to be commutative if for all x and y we have*

$$x \vee y = y \vee x.$$

Ordinary addition, multiplication and matrix addition is commutative, but multiplication of matrices and permutations is not.

DEFINITION A.21 (Identity). *An operation $\cdot$ on the set A is said to have an identity if there exists an element e of A such that for all x we have*

$$e \cdot x = x \cdot e = x.$$

The first thing we notice is that all the example operations above possess an identity, but that ordinary subtraction on the set $\mathbb{R}$ does not possess an identity. The following shows that there can be at most one identity for any given operation.

LEMMA A.22. *If an identity exists then it is unique. It is then called 'the' identity.*

PROOF. Suppose there are two identities e and e'. As e is an identity we have $e \cdot e' = e'$ and as e' is an identity we have $e \cdot e' = e$. Hence, we have $e' = e \cdot e' = e$. □

Usually if we are using an additive notation then we denote the identity by 0 to correspond with the identity for ordinary addition, and if we are using the multiplicative notation then we denote the identity by either 1 or e.

DEFINITION A.23 (Inverses). *Let $+$ be an operation on a set A with identity 0. Let $x \in A$. If there is a $y \in A$ such that*

$$x + y = y + x = 0$$

then we call y an inverse of x.

In the additive notation it is usual to write the inverse of x as $-x$. In the multiplicative notation it is usual to write the inverse as x^{-1}.

All elements in $\mathbb{R}$ have inverses with respect to ordinary addition. All elements in $\mathbb{R}$ except zero have inverses with respect to ordinary multiplication. Every permutation has an inverse with respect to multiplication of permutations. However, only square matrices of non-zero determinant have inverses with respect to matrix multiplication.

The next result shows that an element can have at most one inverse assuming the operation is associative.

LEMMA A.24. *Consider an associative operation on a set A with identity e. Let $x \in A$ have an inverse y, then this inverse is unique, we call it 'the' inverse.*

PROOF. Suppose their are two such inverses y and y' then

$$y = ye = y(xy') = (yx)y' = ey' = y'.$$

Note how we used the associativity property above. □

We shall assume from now on that all operations we shall encounter are associative. Say one wishes to perform the same operation over and over again, for example

$$x \vee x \vee x \vee \cdots \vee x \vee x.$$

If our operation is written additively then we write for $n \in \mathbb{N}$, $n \cdot x$ for $x + \cdots + x$, whilst if our operation is written multiplicatively we write x^n for $x \cdots x$.

The following result can be proved by induction:

LEMMA A.25 (Law of Powers). *For any operation $\circ$ which is associative we have*

$$g^m \circ g^n = g^{m+n}, \ (g^m)^n = g^{m \cdot n}.$$

We can extend the notation to all $n \in \mathbb{Z}$, if x has an inverse (and the operation an identity), by $(-n) \cdot x = n \cdot (-x)$ and $x^{-n} = (x^{-1})^n$.

The following lemma is obvious, but often causes problems as it is slightly counter-intuitive. To get it in your brain consider the case of matrices.

LEMMA A.26. *Consider a set with an associative operation which has an identity, e. If $x, y \in G$ possess inverses then we have*

(1) $(x^{-1})^{-1} = x$.
(2) $(xy)^{-1} = y^{-1}x^{-1}$.

PROOF. For the first we notice

$$x^{-1} \cdot x = e = x \cdot x^{-1}.$$

Hence by definition of inverses the result follows. For the second we have

$$x \cdot y \cdot (y^{-1} \cdot x^{-1}) = x \cdot (y \cdot y^{-1}) \cdot x^{-1} = x \cdot e \cdot x^{-1} = x \cdot x^{-1} = e,$$

and again the result follows by the definition of inverses. □

We have the following dictionary to translate between additive and multiplicative notations:

Additive	Multiplicative
$x + y$	xy
0	1 or e
$-x$	x^{-1}
$n \cdot x$	x^n

6. Groups

DEFINITION A.27. *A group is a set G with a binary operation $\circ$ such that*

(1) $\circ$ *is associative.*
(2) $\circ$ *has an identity element in G.*
(3) *Every element of G has an inverse.*

Note we have not said that the binary operation is closed as this is implicit in our definition of what an operation is. If the operation is also commutative then we say that we have a commutative, or abelian, group.

The following are all groups, as an exercise you should decide on the identity element, what the inverse of each element is, and which groups are abelian.

(1) The integers $\mathbb{Z}$ under addition (written $\mathbb{Z}^+$).
(2) The rationals $\mathbb{Q}$ under addition (written $\mathbb{Q}^+$).

(3) The reals $\mathbb{R}$ under addition (written $\mathbb{R}^+$).
(4) The complexes $\mathbb{C}$ under addition (written $\mathbb{C}^+$).
(5) The rationals (excluding zero) $\mathbb{Q} \setminus \{0\}$ under multiplication (written $\mathbb{Q}^*$).
(6) The reals (excluding zero) $\mathbb{R} \setminus \{0\}$ under multiplication (written $\mathbb{R}^*$).
(7) The complexes (excluding zero) $\mathbb{C} \setminus \{0\}$ under multiplication (written $\mathbb{C}^*$).
(8) The set of n vectors over $\mathbb{Z}, \mathbb{Q}, \ldots$, etc. under vector addition.
(9) The set of $n \times m$ matrices with integer, rational, real or complex entries under matrix addition. This set is written $M_{n\times m}(\mathbb{Z})$, etc. however when $m = n$ we write $M_n(\mathbb{Z})$ instead of $M_{n\times n}(\mathbb{Z})$.
(10) The general linear group (the matrices of non-zero determinant) over the rationals, reals or complexes under matrix multiplication (written $\mathrm{GL}_n(\mathbb{Q})$, etc.).
(11) The special linear group (the matrices of determinant ± 1) over the integers, rationals etc. (written $\mathrm{SL}_n(\mathbb{Z})$, etc.).
(12) The set of permutations on n elements, written S_n and often called the symmetric group on n letters.
(13) The set of continuous (differentiable) functions from $\mathbb{R}$ to $\mathbb{R}$ under pointwise addition.
(14) etc.

The list is endless, a group is one of the most basic concepts in mathematics.

Not all mathematical objects are however groups. Consider the following list of sets and operations which are not groups. You should decide why they are not groups.

(1) The natural numbers $\mathbb{N}$ under ordinary addition or multiplication.
(2) The integers $\mathbb{Z}$ under subtraction or multiplication.

We now give a number of definitions related to groups.

Definition A.28.
The order of a group is the number of elements in the underlying set G and is denoted $|G|$ or $\#G$. The order of an element $g \in G$ is the least positive integer n such that $g^n = e$, if such an n exists otherwise we say that g has infinite order.
A cyclic group G is a group which has an element g such that each element of G can be written in the form g^n for some $n \in \mathbb{Z}$ (in multiplicative notation). If this is the case then one can write $G = \langle g \rangle$ and one says that g is a generator of the group G.

Note, the only element in a group with order one is the identity element and if x is an element of a group then x and x^{-1} have the same order.

Lemma A.29. *If $G = \langle g \rangle$ and g has finite order n then the order of G is n.*

Proof. Every element of G can be written as g^m for some $m \in \mathbb{Z}$, but as g has order n there are only n distinct such values, as

$$g^{n+1} = g^n \circ g = e \circ g = g.$$

So the group G has only n elements. □

Let us relate this back to the permutations which we introduced earlier. Recall that the set of permutations forms a group under composition. It is easy to see that if $\sigma \in S_n$ is a k-cycle then σ has order k in S_n. One can also easily see that if σ is a product of disjoint cycles then the order of σ is the least common multiple of the orders of the constituent cycles.

A subset S of G is said to generate G if every element of G can be written as a product of elements of S. For instance

- the group S_3 is generated by the set $\{(1,2),(1,2,3)\}$,
- the group $\mathbb{Z}^+$ is generated by the element 1,

- the group $\mathbb{Q}^*$ is generated by the set of prime numbers, it therefore has an infinite number of generators.

Note that the order of a group says nothing about the number of generators it has.

An important set of finite groups which are easy to understand is groups obtained by considering the integers modulo a number m. Recall that we have $\mathbb{Z}/m\mathbb{Z} = \{0, 1, \ldots, m-1\}$. This is a group with respect to addition, when we take the non-negative remainder after forming the sum of two elements. It is not a group with respect to multiplication in general, even when we exclude 0. We can, however, get around this by setting

$$(\mathbb{Z}/m\mathbb{Z})^* = \{x \in \mathbb{Z}/m\mathbb{Z} : (m, x) = 1\}.$$

This latter set is a group with respect to multiplication, when we take the non-negative remainder after forming the product of two elements. The order of $(\mathbb{Z}/m\mathbb{Z})^*$ is denoted $\phi(m)$, the Euler ϕ function. This is an important function in the theory of numbers. As an example we have

$$\phi(p) = p - 1,$$

if p is a prime number. We shall return to this function later.

We now turn our attention to subgroups.

DEFINITION A.30. *A subgroup H of a group G is a subset of G which is also a group with respect to the operation of G. We write in this case $H < G$.*

Note that by this definition $\mathrm{GL}_n(\mathbb{R})$ is not a subgroup of $M_n(\mathbb{R})$, although $\mathrm{GL}_n(\mathbb{R}) \subset M_n(\mathbb{R})$. The operation on $\mathrm{GL}_n(\mathbb{R})$ is matrix multiplication whilst that on $M_n(\mathbb{R})$ is matrix addition.

However we do have the subgroup chains:

$$\mathbb{Z}^+ < \mathbb{Q}^+ < \mathbb{R}^+ < \mathbb{C}^+,$$
$$\mathbb{Q}^* < \mathbb{R}^* < \mathbb{C}^*.$$

If we also identify $x \in \mathbb{Z}$ with the diagonal matrix $\mathrm{diag}(x, \ldots, x)$ then we also have that $\mathbb{Z}^+$ is a subgroup of $M_n(\mathbb{Z})$ and so on.

As an important example, consider the set $2\mathbb{Z}$ of even integers, this is a subgroup of $\mathbb{Z}^+$. If we write $\mathbb{Z}^+ = 1\mathbb{Z}$, then we have $n\mathbb{Z} < m\mathbb{Z}$ if and only if m divides n, where

$$m\mathbb{Z} = \{\ldots, -2m, -m, 0, m, 2m, \ldots\}.$$

We hence obtain various chains of subgroups of $\mathbb{Z}^+$,

$$18\mathbb{Z} < 6\mathbb{Z} < 2\mathbb{Z} < \mathbb{Z}^+,$$
$$18\mathbb{Z} < 9\mathbb{Z} < 3\mathbb{Z} < \mathbb{Z}^+,$$
$$18\mathbb{Z} < 6\mathbb{Z} < 3\mathbb{Z} < \mathbb{Z}^+.$$

We now show that these are the only such subgroups of $\mathbb{Z}^+$.

LEMMA A.31. *The only subgroups of $\mathbb{Z}^+$ are $n\mathbb{Z}$ for some positive integer n.*

PROOF. Let H be a subgroup of $\mathbb{Z}^+$. As H is non-empty it must contain an element x and its inverse $-x$. Hence H contains at least one positive element n. Let n denote the least such positive element of H and let m denote an arbitrary non-zero element of H. By Euclidean division, there exist $q, r \in \mathbb{Z}$ with $0 \leq r < n$ such that

$$m = qn + r.$$

Hence $r \in H$, by choice of n this must mean $r = 0$. Therefore all elements of H are of the form nq which is what was required. □

So every subgroup of $\mathbb{Z}^+$ is an infinite cyclic group. This last lemma combined with the earlier subgroup chains gives us a good definition of what a prime number is.

DEFINITION A.32. *A prime number is a (positive) generator of a non-trivial subgroup of* $\mathbb{Z}^+$, *i.e.* $H \neq \mathbb{Z}^+$ *or* 0, *for which no subgroup of* $\mathbb{Z}^+$ *contains* H *except* $\mathbb{Z}^+$ *and* H *itself.*

What is good about this definition is that we have not referred to the multiplicative structure of $\mathbb{Z}$ to define the primes. Also it is obvious that neither zero nor one is a prime number. In addition the above definition allows one to generalize the notion of primality to other settings, for how this is done consult any standard textbook on abstract algebra.

6.1. Normal Subgroups and Cosets. A normal subgroup is particularly important in the theory of groups. The name should not be thought of as meaning that these are the subgroups that normally arise, the name is a historic accident. To define a normal subgroup we first need to define what is meant by conjugate elements.

DEFINITION A.33. *Two elements* x, y *of a group* G *are said to be conjugate if there is an element* $g \in G$ *such that* $x = g^{-1}yg$.

It is obvious that two conjugate elements have the same order. If N is a subgroup of G we define, for any $g \in G$,

$$g^{-1}Ng = \{g^{-1}xg : x \in N\},$$

which is another subgroup of G, called a conjugate of the subgroup N.

DEFINITION A.34. *A subgroup* $N < G$ *is said to be normal if* $g^{-1}Ng \subset N$ *for all* $g \in G$. *If this is the case then we write* $N \lhd G$.

For any group G we have $G \lhd G$ and $\{e\} \lhd G$ and if G is an abelian group then every subgroup of G is normal. The importance of normal subgroups comes from the fact that these are subgroups that we can factor out by. This is related to the cosets of a subgroup which we now go on to introduce.

DEFINITION A.35. *Let* G *be a group and* $H < G$ *(*H *is not necessarily normal). Fix an element* $g \in G$ *then we define the left coset of* H *with respect to* g *to be the set*

$$gH = \{gh : h \in H\}.$$

Similarly we define the right coset of H *with respect to* g *to be the set*

$$Hg = \{hg : h \in H\}.$$

Let H denote a subgroup of G then one can show that the set of all left (or right) cosets of H in G forms a partition of G, but we leave this to the reader. In addition if $a, b \in G$ then $aH = bH$ if and only if $a \in bH$, which is also equivalent to $b \in aH$, a fact which we also leave to the reader to show. Note that we can have two equal cosets $aH = bH$ without having $a = b$.

What these latter facts show is that if we define the relation R_H on the group G with respect to the subgroup H by

$$(a, b) \in R_H \text{ if and only if } a = bh \text{ for some } h \in H,$$

then this relation is an equivalence relation. The equivalence classes are just the left cosets of H in G.

The number of left cosets of a subgroup H in G we denote by $(G : H)_L$, the number of right cosets we denote by $(G : H)_R$. We are now in a position to prove the most important theorem of elementary group theory, namely Lagrange's Theorem.

THEOREM A.36 (Lagrange's Theorem). *Let* H *be a subgroup of a finite group* G *then*

$$|G| = (G : H)_L \cdot |H|$$
$$(G : H)_R \cdot |H|.$$

Before we prove this result we state some obvious important corollaries;

COROLLARY A.37.
We have $(G:H)_L = (G:H)_R$; this common number we denote by $(G:H)$ and call it the index of the subgroup H in G.
The order of a subgroup and the index of a subgroup both divide the order of the group.
If G is a group of prime order, then G has only the subgroups G and $\langle e \rangle$.

We now return to the proof of Lagrange's Theorem.

PROOF. We form the following collection of distinct left cosets of H in G which we define inductively. Put $g_1 = e$ and assume we are given i cosets by $g_1H, \ldots, g_iH$. Now take an element g_{i+1} not lying in any of the left cosets g_jH for $j \leq i$. After a finite number of such steps we have exhausted elements in the group G. So we have disjoint union of left cosets which cover the whole group.

$$G = \bigcup_{1 \leq i \leq (G:H)_L} g_iH.$$

We also have for each i, j that $|g_iH| = |g_jH|$, this follows from the fact that the map

$$\begin{aligned} H &\longrightarrow gH \\ h &\longmapsto gh \end{aligned}$$

is a bijective map of sets. Hence

$$|G| = \sum_{1 \leq i \leq (G:H)_L} |g_iH| = (G:H)_L|H|.$$

The other equality follows using the same argument. □

We can also deduce from the corollaries the following

LEMMA A.38. *If G is a group of prime order then it is cyclic.*

PROOF. If $g \in G$ is not the identity then $\langle g \rangle$ is a subgroup of G of order ≥ 2. But then it must have order $|G|$ and so G is cyclic. □

We can use Lagrange's Theorem to write down the subgroups of some small groups. For example, consider the group S_3 this has order 6 so by Lagrange's Theorem its subgroups must have order $1, 2, 3$ or 6. It is easy to see that the only subgroups are therefore:

- one subgroup of order 1; namely $\langle (1) \rangle$,
- three subgroups of order 2; namely $\langle (1,2) \rangle$, $\langle (1,3) \rangle$ and $\langle (2,3) \rangle$,
- one subgroup of order 3; namely $\langle (1,2,3) \rangle$,
- one subgroup of order 6, which is S_3 obviously.

6.2. Factor or Quotient Groups. Throughout this subsection let G be a group with a normal subgroup N. The following elementary lemma, whose proof we again leave to the reader, gives us our justification for looking at normal subgroups.

LEMMA A.39. *Let $H < G$ then the following are equivalent:*

(1) *$xH = Hx$ for all $x \in G$.*
(2) *$x^{-1}Hx = H$ for all $x \in G$.*
(3) *$H \lhd G$.*
(4) *$x^{-1}hx \in H$ for all $x \in G$ and $h \in H$.*

By G/N we denote the set of left cosets of N, note that these are the same as the right cosets of N. We note that two cosets, g_1N and g_2N are equal if and only if $g_1^{-1}g_2 \in N$.

We wish to turn G/N into a group, the so-called factor group or quotient group. Let g_1N and g_2N denote any two elements of G/N then we define the product of their left cosets to be $(g_1g_2)N$.

We first need to show that this is a well-defined operation, i.e. if we replace g_1 by g_1' and g_2 by g_2' with $g_1^{-1}g_1' = n_1 \in N$ and $g_2^{-1}g_2' = n_2 \in N$ then our product still gives the same coset. In other words we wish to show

$$(g_1g_2)N = (g_1'g_2')N.$$

Now let $x \in (g_1g_2)N$ then $x = g_1g_2n$ for some $n \in N$. Then $x = g_1'n_1^{-1}g_2'n_2^{-1}n$. But as G is normal (left cosets = right cosets) we have ${n_1'}^{-1}g_2' = g_2'n_3$ for some $n_3 \in N$. Hence

$$x = g_1'g_2'n_3n_2^{-1}n \in g_1'g_2'N.$$

This proves the first inclusion, the other follows similarly. We conclude that our operation on G/N is well defined. One can also show that if N is an arbitrary subgroup of G and we define the operation on the cosets above then this is only a well-defined operation if N is a normal subgroup of G.

So we have a well-defined operation on G/N, we now need to show that this operation satisfies the axioms of a group:

- As an identity we take $eN = N$, since for all $g \in G$ we have
$$eN \cdot gN = (eg)N = gN.$$
- As an inverse of (gN) we take $g^{-1}N$ as
$$gN.g^{-1}N = (gg^{-1})N = eN = N.$$
- Associativity follows from
$$\begin{aligned}(g_1N)(g_2N \cdot g_3N) &= g_1N((g_2g_3)N) = (g_1(g_2g_3))N \\ &= ((g_1g_2)g_3)N = ((g_1g_2)N)g_3N \\ &= (g_1N \cdot g_2N)(g_3N).\end{aligned}$$

We now present some examples.

(1) Let G be an arbitrary finite group of order greater than one, let H be a subgroup of G. Then $H = G$ and $H = \{e\}$ are always normal subgroups of G.
(2) If $H = G$ then there is only one coset and so we have $G/G = \{G\}$ is a group of order one.
(3) If $H = \{e\}$ then the cosets of H are the one-element subsets of G. That is $G/\{e\} = \{\{g\} : g \in G\}$.
(4) Put $G = S_3$ and $N = \{(1), (1,2,3), (1,3,2)\}$, then N is a normal subgroup of G. The cosets of N in G are N and $(1,2)N$ with
$$((1,2)N)^2 = (1,2)^2N = (1)N = N.$$
Hence $S_3/\langle(1,2,3)\rangle$ is a cyclic group of order 2.
(5) If G is abelian then every subgroup H of G is normal, so one can always form the quotient group G/H.
(6) Since $(\mathbb{Z}, +)$ is abelian we have that $m\mathbb{Z}$ is always a normal subgroup. Forming the quotient group $\mathbb{Z}/m\mathbb{Z}$ we obtain the group of integers modulo m under addition.

6.3. Homomorphisms. Let G_1 and G_2 be two groups, we wish to look at the functions from G_1 to G_2. Obviously one could look at all such functions, however by doing this we would lose all the structure that the group laws give us. We restrict ourselves to maps which preserve this group law.

DEFINITION A.40. *A homomorphism from a group G_1 to a group G_2 is a function f with domain G_1 and codomain G_2 such that for all $x, y \in G_1$ we have*

$$f(x \cdot y) = f(x) \cdot f(y).$$

Note multiplication on the left is with the operation of the group G_1 whilst the multiplication on the right is with respect to the operation of G_2. As examples we have

(1) The identity map $\mathrm{id}_G : G \to G$, where $\mathrm{id}_G(g) = g$ is a group homomorphism.
(2) Consider the function $\mathbb{R}^+ \to \mathbb{R}^*$ given by $f(x) = e^x$. This is a homomorphism as for all $x, y \in \mathbb{R}$ we have
$$e^{x+y} = e^x e^y.$$
(3) Consider the map from $\mathbb{C}^*$ to $\mathbb{R}^*$ given by $f(z) = |z|$. This is also a homomorphism.
(4) Consider the map from $\mathrm{GL}_n(\mathbb{C})$ to $\mathbb{C}^*$ given by $f(A) = \det(A)$, this is a group homomorphism as $\det(AB) = \det(A) \cdot \det(B)$ for any two elements of $\mathrm{GL}_n(\mathbb{C})$.

Two elementary properties of homomorphisms are summarized in the following lemma.

LEMMA A.41. *Let $f : G_1 \to G_2$ be a homomorphism of groups, then*

(1) $f(e_1) = e_2$.
(2) *For all $x \in G_1$ we have $f(x^{-1}) = (f(x))^{-1}$.*

PROOF. For the first result we have $e_2 f(x) = f(x) = f(e_1 x) = f(e_1)f(x)$ and so
$$e_2 = f(x)f(x)^{-1} = f(e_1)f(x)f(x)^{-1} = f(e_1)$$
as required.

Now for the second we have
$$f(x^{-1})f(x) = f(x^{-1}x) = f(e_1) = e_2,$$
so the result follows by definition. □

For any homomorphism f from G_1 to G_2 there are two special subgroups associated with f.

DEFINITION A.42.
The kernel of f is the set
$$\mathrm{Ker} f = \{x \in G_1 : f(x) = e_2\}.$$
The image of f is the set
$$\mathrm{Im} f = \{y \in G_2 : y = f(x),\ x \in G_1\}.$$

LEMMA A.43. *$\mathrm{Ker} f$ is a normal subgroup of G_1.*

PROOF. We first show that it is a subgroup. It is certainly non-empty as $e_1 \in \mathrm{Ker} f$ as $f(e_1) = e_2$. Now if $x \in \mathrm{Ker} f$ then $f(x^{-1}) = f(x)^{-1} = e_2^{-1} = e_2$, hence $x^{-1} \in \mathrm{Ker} f$. Hence to show that $\mathrm{Ker} f$ is a subgroup we only have to show that for all $x, y \in \mathrm{Ker} f$ we have $xy^{-1} \in \mathrm{Ker} f$. But this is easy as if $x, y \in \mathrm{Ker} f$ then we have
$$f(xy^{-1}) = f(x)f(y^{-1}) = e_2 e_2 = e_2,$$
and we are done.

We now show that $\mathrm{Ker} f$ is in fact a normal subgroup of G_1. We need to show that if $x \in \mathrm{Ker} f$ then $g^{-1}xg \in \mathrm{Ker} f$ for all $g \in G_1$. So let $x \in \mathrm{Ker} f$ and let $g \in G_1$, then we have
$$f(g^{-1}xg) = f(g^{-1})f(x)f(g) = f(g)^{-1}e_2 f(g) = f(g)^{-1}f(g) = e_2,$$
so we are done. □

LEMMA A.44. *$\mathrm{Im} f$ is a subgroup of G_2.*

PROOF. $\mathrm{Im} f$ is certainly non-empty as $f(e_1) = e_2$. Now suppose $y \in \mathrm{Im} f$ so there is an $x \in G_2$ such that $f(x) = y$, then $y^{-1} = f(x)^{-1} = f(x^{-1})$ and $x^{-1} \in G_1$ so $y^{-1} \in \mathrm{Im} f$.

Now suppose $y_1, y_2 \in \mathrm{Im} f$, hence for some x_1, x_2 we have
$$y_1 y_2^{-1} = f(x_1)f(x_2^{-1}) = f(x_1 x_2^{-1}).$$
Hence $\mathrm{Im} f < G_2$. □

It is clear that $\mathrm{Im}f$ in some sense measures whether the homomorphism f is surjective as f is surjective if and only if $\mathrm{Im}f = G_2$. Actually the set $G_2/\mathrm{Im}f$ is a better measure of the surjectivity of the function. What we also have is that $\mathrm{Ker}f$ measures how far from injective f is, due to the following result.

LEMMA A.45. *A homomorphism, f, is injective if and only if* $\mathrm{Ker}f = \{e_1\}$.

PROOF. Assume f is injective, then we know that if $f(x) = e_2 = f(e_1)$ then $x = e_1$ and so $\mathrm{Ker}f = \{e_1\}$.

Now assume that $\mathrm{Ker}f = \{e_1\}$ and let $x, y \in G_1$ be such that $f(x) = f(y)$. Then

$$f(xy^{-1}) = f(x)f(y^{-1}) = f(x)f(y)^{-1} = f(x)f(x)^{-1} = e_2.$$

So $xy^{-1} \in \mathrm{Ker}f$, but then $xy^{-1} = e_1$ and so $x = y$. So f is injective. □

Bijective homomorphisms allow us to categorize groups more effectively, as the following definition elaborates.

DEFINITION A.46. *A homomorphism f is said to be an isomorphism if it is bijective. Two groups are said to be isomorphic if there is an isomorphism between them, in which case we write $G_1 \cong G_2$.*

Note this means that isomorphic groups have the same number of elements. Indeed for all intents and purposes one may as well assume that isomorphic groups are equal, since they look the same up to relabelling of elements.

Isomorphisms satisfy the following properties,

- If $f : G_1 \to G_2$ and $g : G_2 \to G_3$ are isomorphisms then $g \circ f$ is also an isomorphism, i.e. isomorphisms are transitive.
- If $f : G_1 \to G_2$ is an isomorphism then so is $f^{-1} : G_2 \to G_1$, i.e. isomorphisms are symmetric.

From this we see that the relation 'is isomorphic to' is an equivalence relation on the class of all groups. This justifies our notion of isomorphic being like equal.

Let G_1, G_2 be two groups, then we define the product group $G_1 \times G_2$ to be the set $G_1 \times G_2$ of ordered pairs (g_1, g_2) with $g_1 \in G_1$ and $g_2 \in G_2$. The group operation on $G_1 \times G_2$ is given componentwise:

$$(g_1, g_2) \circ (g_1', g_2') = (g_1 \circ g_1', g_2 \circ g_2').$$

The first $\circ$ refers to the group $G_1 \times G_2$, the second to the group G_1 and the third to the group G_2. Some well-known groups can actually be represented as product groups. For example, consider the map

$$\begin{aligned} \mathbb{C}^+ &\longrightarrow \mathbb{R}^+ \times \mathbb{R}^+ \\ z &\longmapsto (\mathrm{Re}(z), \mathrm{Im}(z)). \end{aligned}$$

This map is obviously a bijective homomorphism, hence we have $\mathbb{C}^+ \cong \mathbb{R}^+ \times \mathbb{R}^+$.

We now come to a crucial theorem which says that the concept of a quotient group is virtually equivalent to the concept of a homomorphic image.

THEOREM A.47 (First Isomorphism Theorem for Groups). *Let f be a homomorphism from a group G_1 to a group G_2. Then*

$$G_1/\mathrm{Ker}f \cong \mathrm{Im}f.$$

The proof of this result can be found in any introductory text on abstract algebra. Note that $G_1/\mathrm{Ker}f$ makes sense as $\mathrm{Ker}f$ is a normal subgroup of G.

7. Rings

A ring is an additive finite abelian group with an extra operations, usually denoted by multiplication, such that the multiplication operation is associative and has an identity element. The addition and multiplication operations are linked via the distributive law,

$$a \cdot (b + c) = a \cdot b + a \cdot c = (b + c) \cdot a.$$

If the multiplication operation is commutative then we say we have a commutative ring.

The following are examples of rings.

- integers under addition and multiplication,
- polynomials with coefficients in $\mathbb{Z}$, denoted $\mathbb{Z}[X]$,
- integers modulo a number m, denoted $\mathbb{Z}/m\mathbb{Z}$.

Although one can consider subrings they turn out to be not so interesting. Of more interest are the ideals of the ring, these are additive subgroups $I < R$ such that

$$i \in I \text{ and } r \in R \text{ implies } i \cdot r \in I.$$

Examples of ideals in a ring are the principal ideals which are those additive subgroups generated by a single ring element. For example if $R = \mathbb{Z}$ then the principal ideals are the ideals $m\mathbb{Z}$, for each integer m.

Just as with normal subgroups and groups, where we formed the quotient group, we can with ideals and rings form the quotient ring. If we take $R = \mathbb{Z}$ and $I = m\mathbb{Z}$ for some integer m then the quotient ring is the ring $\mathbb{Z}/m\mathbb{Z}$ of integers modulo m under addition and multiplication modulo m. This leads us naturally to the Chinese Remainder Theorem.

THEOREM A.48 (CRT). *Let $m = p_1^{z_1} \dots p_t^{z_t}$ be the prime factorization of m, then the following map is a ring isomorphism*

$$f : \begin{array}{ccc} \mathbb{Z}/m\mathbb{Z} & \longrightarrow & \mathbb{Z}/p_1^{z_1}\mathbb{Z} \times \cdots \times \mathbb{Z}/p_t^{z_t}\mathbb{Z} \\ x & \longmapsto & (x \pmod{p_1^{z_1}}, \dots, x \pmod{p_t^{z_t}}). \end{array}$$

PROOF. This can be proved by induction on the number of prime factors of m. We leave the details to the interested reader. □

We shall now return to the Euler ϕ function mentioned earlier. Remember $\phi(n)$ denotes the order of the group $\mathbb{Z}/n\mathbb{Z}^*$. We would like to be able to calculate this value easily.

LEMMA A.49. *Let $m = p_1^{z_1} \dots p_t^{z_t}$ be the prime factorization of m. Then we have*

$$\phi(m) = \phi(p_1^{z_1}) \dots \phi(p_t^{z_t}).$$

PROOF. This follows from the Chinese Remainder Theorem, as the ring isomorphism

$$\mathbb{Z}/m\mathbb{Z} \cong \mathbb{Z}/p_1^{z_1}\mathbb{Z} \times \cdots \times \mathbb{Z}/p_t^{z_t}\mathbb{Z}$$

induces a group isomorphism

$$(\mathbb{Z}/m\mathbb{Z})^* \cong (\mathbb{Z}/p_1^{z_1}\mathbb{Z})^* \times \cdots \times (\mathbb{Z}/p_t^{z_t}\mathbb{Z})^*.$$

□

To compute the Euler ϕ function all we now require is:

LEMMA A.50. *Let p be a prime number, then $\phi(p^e) = p^{e-1}(p-1)$.*

PROOF. There are $p^e - 1$ elements of $\mathbb{Z}$ satisfying $1 \leq k < p^e$, of these we must eliminate those of the form $k = rp$ for some r. But $1 \leq rp < p^e$ implies $1 \leq r < p^{e-1}$, hence there are $p^{e-1} - 1$ possible values of r. So we obtain

$$\phi(p^e) = (p^e - 1) - (p^{e-1} - 1)$$

from which the result follows. □

An ideal I of a ring is called prime if $x \cdot y \in I$ implies either $x \in I$ or $y \in I$. Notice, the ideals $I = m\mathbb{Z}$ of the ring $\mathbb{Z}$ are prime if and only if m is plus or minus a prime number.

The prime ideals are special as if we take the quotient of a ring by a prime ideal then we obtain a field. Hence, $\mathbb{Z}/p\mathbb{Z}$ is a field. This brings us naturally on to the subject of fields.

8. Fields

A field is essentially two abelian groups stuck together using the distributive law. More formally:

DEFINITION A.51. *A field is an additive abelian group F, such that $F \setminus \{0\}$ also forms an abelian group with respect to another operation (which is usually written multiplicatively). The two operations, addition and multiplication, are linked via the distributive law:*

$$a \cdot (b + c) = a \cdot b + a \cdot c = (b + c) \cdot a.$$

Many fields that one encounters have infinitely many elements. Every finite field either contains $\mathbb{Q}$ as a subfield, in which case we say it has characteristic zero, or it contains $\mathbb{F}_p$ as a subfield in which case we say it has characteristic p. The only fields with finitely many elements have p^r elements when p is a prime. We denote such fields by $\mathbb{F}_{p^r}$, for each value of r there is only one such field up to isomorphism. Such finite fields are often called Galois fields.

Let F be a field, we denote by $F[X]$ the ring of polynomials in a single variable X with coefficients in the field F. The set $F(X)$ of rational functions in X is the set of functions of the form

$$f(X)/g(X),$$

where $f(X), g(X) \in F[X]$ and $g(X)$ is not the zero polynomial. The set $F(X)$ is a field with respect to the obvious addition and multiplication. One should note the difference in the notation of the brackets, $F[X]$ and $F(X)$.

Let f be a polynomial of degree n with coefficients in $\mathbb{F}_p$ which is irreducible. Let θ denote a root of f. Consider the set

$$\mathbb{F}_p(\theta) = \{a_0 + a_1\theta + \cdots + a_{n-1}\theta^{n-1} : a_i \in \mathbb{F}_p\}.$$

Given two elements of $\mathbb{F}_p(\theta)$ one adds them componentwise and multiplies them as polynomials in θ but then one takes the remainder of the result on division by $f(\theta)$. The set $\mathbb{F}_p(\theta)$ is a field, there are field-theoretic isomorphisms

$$\mathbb{F}_{p^n} = \mathbb{F}_p(\theta) = \mathbb{F}_p[X]/(f),$$

where (f) represents the ideal

$$\{f \cdot g : g \in \mathbb{F}_p[X]\}.$$

To be more concrete let us look at the specific example given by choosing a value of $p \equiv 3 \pmod 4$ and $f(X) = X^2 + 1$. Now since $p \equiv 3 \pmod 4$ the polynomial f is irreducible over $\mathbb{F}_p[X]$ and so the quotient $\mathbb{F}_p[X]/(f)$ forms a field, which is isomorphic to $\mathbb{F}_{p^2}$.

Let i denote a root of the polynomial $X^2 + 1$. The field $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ consists of numbers of the form

$$a + bi$$

where a and b are integers modulo p. We add such numbers as

$$(a + bi) + (c + di) = (a + c) + (b + d)i.$$

We multiply such numbers as

$$(a + bi)(c + di) = (ac + (ad + bc)i + bdi^2) = (ac - bd) + (ad + bc)i.$$

Here is another example. Let θ denote a root of the polynomial $x^3 + 2$, then an element of

$$\mathbb{F}_{7^3} = \mathbb{F}_7(\theta)$$

can be represented by

$$a + b\theta + c\theta^2.$$

Multiplication of two such elements gives

$$\begin{aligned}(a + b\theta + c\theta^2)(a' + b'\theta + c'\theta^2) &= aa' + \theta(a'b + b'a) + \theta^2(ac' + bb' + ca') \\ &\quad + \theta^3(bc' + cb') + cc'\theta^4 \\ &= (aa' - 2bc' - 2cb') + \theta(a'b + b'a - 2cc') \\ &\quad + \theta^2(ac' + bb' + ca').\end{aligned}$$

9. Vector Spaces

DEFINITION A.52. *Given a field K a vector space (or a K-vector space) V is an abelian group (also denoted V) and an external operation $K \times V \to V$ (called scalar multiplication) which satisfies the following axioms: For all $\lambda, \mu \in K$ and all $x, y \in V$ we have*

(1) $\lambda(\mu x) = (\lambda\mu)x$.
(2) $(\lambda + \mu)x = \lambda x + \mu x$.
(3) $1_K x = x$.
(4) $\lambda(x + y) = \lambda x + \lambda y$.

One often calls the elements of V the vectors and the elements of K the scalars. Note that we are not allowed to multiply or divide two vectors. We shall start with some examples:

- For a given field K and an integer $n \geq 1$, let $V = K^n = K \times \cdots \times K$ be the n-fold Cartesian product. This is a vector space over K with respect to the usual addition of vectors and multiplication by scalars. A special case of $n = 1$ shows that any field is a vector space over itself. When $K = \mathbb{R}$ and $n = 2$ we obtain the familiar system of geometric vectors in the plane. When $n = 3$ and $K = \mathbb{R}$ we obtain 3-dimensional vectors. Hence you can already see the power of vector spaces as they allow us to consider n-dimensional space in a concrete way.
- Let K be a field and consider the set of polynomials over K, namely $K[X]$. This is a vector space with respect to addition of polynomials and multiplication by elements of K.
- Let K be a field and E any set at all. Define V to be the set of functions $f : E \to K$. Given $f, g \in V$ and $\lambda \in K$ one can define the sum $f + g$ and scalar product λf via

  $$(f + g)(x) = f(x) + g(x) \text{ and } (\lambda f)(x) = \lambda f(x).$$

  We leave the reader the simple task to check that this is a vector space.
- The set of all continuous functions $f : \mathbb{R} \to \mathbb{R}$ is a vector space over $\mathbb{R}$. This follows from the fact that if f and g are continuous then so is $f + g$ and λf for any $\lambda \in \mathbb{R}$. Similarly the set of all differentiable functions $f : \mathbb{R} \to \mathbb{R}$ also forms a vector space.

9.1. Vector Sub-spaces. Let V be a K-vector space and let W be a subset of V. W is said to be a vector subspace (or just subspace) of V if

(1) W is a subgroup of V with respect to addition.
(2) W is closed under scalar multiplication.

By this last condition we mean $\lambda x \in W$ for all $x \in W$ and all $\lambda \in K$. What this means is that a vector subspace is a subset of V which is also a vector space with respect to the same addition and multiplication laws as are on V. There are always two trivial subspaces of a space, namely $\{0\}$ and V itself. Here are some more examples:

- Let $V = K^n$ and $W = \{(\xi_1, \ldots, \xi_n) \in K^n : \xi_n = 0\}$.
- Let $V = K^n$ and $W = \{(\xi_1, \ldots, \xi_n) \in K^n : \xi_1 + \cdots + \xi_n = 0\}$.
- $V = K[X]$ and $W = \{f \in K[X] : f = 0 \text{ or } \deg f \leq 10\}$.

- $\mathbb{C}$ is a natural vector space over $\mathbb{Q}$, and $\mathbb{R}$ is a vector subspace of $\mathbb{C}$.
- Let V denote the set of all continuous functions from $\mathbb{R}$ to $\mathbb{R}$ and W the set of all differentiable functions from $\mathbb{R}$ to $\mathbb{R}$. Then W is a vector subspace of V.

9.2. Properties of Elements of Vector Spaces. Before we go any further we need to define certain properties which sets of elements of vector spaces can possess. For the following definitions let V be a K-vector space and let $x_1, \ldots, x_n$ and x denote elements of V.

DEFINITION A.53.
x is said to be a linear combination of $x_1, \ldots, x_n$ if there exists scalars $\lambda_i \in K$ such that

$$x = \lambda_1 x_1 + \cdots + \lambda_n x_n.$$

The elements $x_1, \ldots, x_n$ are said to be linearly independent if the relation

$$\lambda_1 x_1 + \cdots + \lambda_n x_n = 0$$

implies that $\lambda_1 = \cdots = \lambda_n = 0$. If $x_1, \ldots, x_n$ are not linearly independent then they are said to be linearly dependent.
A subset A of a vector space is linearly independent or free if whenever $x_1, \ldots, x_n$ are finitely many elements of A, they are linearly independent.
A subset A of a vector space V is said to span (or generate) V if every element of V is a linear combination of finitely many elements from A.
If there exists a finite set of vectors spanning V then we say that V is finite-dimensional.

We now give some examples of the last concept.

- The vector space $V = K^n$ is finite-dimensional. For let

  $$e_i = (0, \ldots, 0, 1, 0, \ldots, 0)$$

  be the n-tuple with 1 in the ith-place and 0 elsewhere. Then V is spanned by the vectors $e_1, \ldots, e_n$. Note the analogy with the geometric plane.
- $\mathbb{C}$ is a finite-dimensional vector space over $\mathbb{R}$, and $\{1, \sqrt{-1}\}$ is a spanning set.
- $\mathbb{R}$ and $\mathbb{C}$ are not finite-dimensional vector spaces over $\mathbb{Q}$. This is obvious since $\mathbb{Q}$ has countably many elements, any finite-dimensional subspace over $\mathbb{Q}$ will also have countably many elements. However it is a basic result in analysis that both $\mathbb{R}$ and $\mathbb{C}$ have uncountably many elements.

Now some examples about linear independence:

- In the vector space $V = K^n$ the n-vectors $e_1, \ldots, e_n$ defined earlier are linearly independent.
- In the vector space $\mathbb{R}^3$ the vectors $x_1 = (1, 2, 3)$, $x_2 = (-1, 0, 4)$ and $x_3 = (2, 5, -1)$ are linearly independent.
- On the other hand, the vectors $y_1 = (2, 4, -3)$, $y_2 = (1, 1, 2)$ and $y_3 = (2, 8, -17)$ are linearly dependent as we have $3y_1 - 4y_2 - y_3 = 0$.
- In the vector space (and ring) $K[X]$ over the field K the infinite set of vectors

  $$\{1, X, X^2, X^3, \ldots\}$$

  is linearly independent.

9.3. Dimension and Bases.

DEFINITION A.54. *A subset A of a vector space V which is linearly independent and spans the whole of V is called a basis.*

Given a basis then each element in V can be written in a unique way: for if $x_1, \ldots, x_n$ is a basis and suppose that we can write x as a linear combination of the x_i in two ways i.e. $x = \lambda_1 x_1 + \cdots + \lambda_n x_n$ and $x = \mu_1 x_1 + \cdots + \mu_n x_n$. Then we have

$$0 = x - x = (\lambda_1 - \mu_1)x_1 + \cdots + (\lambda_n - \mu_n)x_n$$

and as the x_i are linearly independent we obtain $\lambda_i - \mu_i = 0$, i.e. $\lambda_i = \mu_i$.

We have the following examples.

- The vectors $e_1, \ldots, e_n$ of K^n introduced earlier form a basis of K^n. This basis is called the standard basis of K^n.
- The set $\{1, i\}$ is a basis of the vector space $\mathbb{C}$ over $\mathbb{R}$.
- The infinite set $\{1, X, X^2, X^2, \ldots\}$ is a basis of the vector space $K[X]$.

By way of terminology we call the vector space $V = \{0\}$ the trivial or zero vector space. All other vector spaces are called non-zero. To make the statements of the following theorems easier we shall say that the zero vector space has the basis set $\emptyset$.

THEOREM A.55. *Let V be a finite-dimensional vector space over a field K. Let C be a finite subset of V which spans V and let A be a subset of C which is linearly independent. Then V has a basis, B, such that $A \subset B \subset C$.*

PROOF. We can assume that V is non-zero. Consider the collection of all subsets of C which are linearly independent and contain A. Certainly such subsets exist since A is itself an example. So choose one such subset B with as many elements as possible. By construction B is linearly independent. We now show that B spans V.

Since C spans V we only have to show that every element $x \in C$ is a linear combination of elements of B. This is trivial when $x \in B$ so assume that $x \notin B$. Then $B' = B \cup \{x\}$ is a subset of C larger than B, whence B' is linearly dependent, by choice of B. If $x_1, \ldots, x_r$ are the distinct elements of B this means that there is a linear relation

$$\lambda_1 x_1 + \cdots + \lambda_r x_r + \lambda x = 0,$$

in which not all the scalars, λ_i, λ, are zero. In fact $\lambda \neq 0$. So we may rearrange to express x as a linear combination of elements of B, as λ has an inverse in K. □

COROLLARY A.56. *Every finite-dimensional vector space, V, has a basis.*

PROOF. We can assume that V is non-zero. Let C denote a finite spanning set of V and let $A = \emptyset$ and then apply the above theorem. □

The last theorem and its corollary are true if we drop the assumption of finite dimensional. However then we require much more deep machinery to prove the result. The following result is crucial to the study of vector spaces as it allows us to define the dimension of a vector space. One should think of dimension of a vector space as the same as dimension of the 2-D or 3-D space one is used to.

THEOREM A.57. *Suppose a vector space V contains a spanning set of m elements and a linearly independent set of n elements. Then $m \geq n$.*

PROOF. Let $A = \{x_1, \ldots, x_m\}$ span V, and let $B = \{y_1, \ldots, y_n\}$ be linearly independent and suppose that $m < n$. Hence we wish to derive a contradiction.

We successively replace the xs by the ys, as follows. Since A spans V, there exists scalars $\lambda_1, \ldots, \lambda_m$ such that

$$y_1 = \lambda_1 x_1 + \cdots + \lambda_m x_m.$$

At least one of the scalars, say λ_1, is non-zero and we may express x_1 in terms of y_1 and $x_2, \ldots, x_m$. It is then clear that $A_1 = \{y_1, x_2, \ldots, x_m\}$ spans V.

We repeat the process m times and conclude that $A_m = \{y_1, \ldots, y_m\}$ spans V. (One can formally dress this up as induction if one wants to be precise, which we will not bother with.)

By hypothesis $m < n$ and so A_m is not the whole of B and y_{m+1} is a linear combination of $y_1, \ldots, y_m$, as A_m spans V. This contradicts the fact that B is linearly independent. □

Let V be a finite-dimensional vector space. Suppose A is a basis of m elements and B a basis of n elements. By applying the above theorem twice (once to A and B and once to B and A) we deduce that $m = n$. From this we conclude the following theorem.

THEOREM A.58. *Let V be a finite-dimensional vector space. Then all bases of V have the same number of elements, we call this number the dimension of V (written $\dim V$).*

It is clear that $\dim K^n = n$. This agrees with our intuition that a vector with n components lives in an n-dimensional world, and that $\dim \mathbb{R}^3 = 3$. Note when referring to dimension we sometimes need to be clear about the field of scalars. If we wish to emphasise the field of scalars we write $\dim_K V$. This can be important, for example if we consider the complex numbers we have

$$\dim_{\mathbb{C}} \mathbb{C} = 1, \ \dim_{\mathbb{R}} \mathbb{C} = 2, \ \dim_{\mathbb{Q}} \mathbb{C} = \infty.$$

The following results are left as exercises.

THEOREM A.59. *If V is a (non-zero) finite-dimensional vector space, of dimension n, then*

(1) *Given any linearly independent subset A of V, there exists a basis B such that $A \subset B$.*
(2) *Given any spanning set C of V, there exists a basis B such that $B \subset C$.*
(3) *Every linearly independent set in V has $\leq n$ elements.*
(4) *If a linearly independent set has exactly n elements then it is a basis.*
(5) *Every spanning set has $\geq n$ elements.*
(6) *If a spanning set has exactly n elements then it is a basis.*

THEOREM A.60. *Let W be a subspace of a finite-dimensional vector space V. Then $\dim W \leq \dim V$, with equality holding if and only if $W = V$.*

Index

A5/1, 118, 119
A5/2, 118
Abadi, 148
abelian group, 4, 23, 26, 28, 201, 405, 410–412
access structure, 347–349
active attack, 91
adaptive chosen ciphertext attack, 289–291
Adleman, 187, 197, 219
Adleman–Huang algorithm, 187, 310
AES, 124, 131
affine point, 24
algebraic normal form, 115
alternating step generator, 117
anomalous, 29, 214
ANSI, 126
anti-symmetric, 396
arithmetic circuit, 384, 388
ASN.1, 261
associative, 4, 401, 402, 410
asymmetric cryptosystems, 37
Atlantic City algorithm, 309
authenticated key exchange, 229
automorphism, 9
avalanche effect, 156

Baby-Step/Giant-Step, 204–207, 224
BAN logic, 148–151
basis, 274, 275, 413
basis matrix, 274, 275
Baudot code, 96–98, 106
Bayes' Theorem, 20, 82
Bellare, 322, 333
Berlekamp–Massey algorithm, 115
Berlekamp–Welch algorithm, 355–357
Bertrand, 57, 61
bigrams, 38
bijective, 398, 409
binary circuit, 384, 385, 388
binary Euclidean algorithm, 12, 244, 246
binary exponentiation, 234, 235
binding, 256, 362–364
birthday paradox, 21, 154, 207
bit security, 306
block cipher, 41, 44, 123, 125, 131, 134–137, 161
Bombe, 56, 64–68, 70–72
Boneh, 265, 281, 282
BPP, 310
Burrows, 148

CA, 256, 260
Caesar cipher, 39
Carmichael numbers, 185
cartesian product, 395
CBC Mode, 134, 136, 138, 161, 328, 329
CBC-MAC, 161
CCA, 291, 328
CCA1, 289
CCA2, 289, 291
certificate authority, 256, 264, 358
certificate chains, 258
Certificate Revocation List, 258
CFB Mode, 135, 137, 138
characteristic, 7, 9, 24, 26, 29–33, 245, 246, 411
Chaum–Pedersen protocol, 375
Chinese Remainder Theorem, 3, 10, 13, 14, 18, 22, 171, 177, 181, 201, 202, 204, 238, 249, 280, 283, 284, 294, 410
chord-tangent process, 25, 26
chosen ciphertext attack, 91, 173, 179, 289, 293
chosen plaintext attack, 47, 91, 173, 179, 289
cillies, 58
cipher, 37
ciphertext, 37
closed, 4
co-NP, 301
co-P, 300, 301
co-RP, 310
Cocks, 167
coding theory, 351
codomain, 315, 397
collision resistant, 154, 155
Colossus, 106, 107
commitment, 362
commitment scheme, 362–365, 377
commutative, 4, 401, 402, 410
commutative ring, 410
completeness, 371
compression function, 155, 156
computational zero-knowledge, 371
computationally secure, 77, 362
concealing, 362–364
conditional entropy, 86

conditional probability, 20, 79, 86
conjugate, 405
connection polynomial, 110
continued fraction, 271–273, 277
Coppersmith, 277, 281
Coppersmith's Theorem, 279–283
coprime, 6
correlation attack, 116, 117
coset, 405, 406
Couveignes, 198
CPA, 289, 328
Cramer, 330, 331, 335, 339, 340, 376
Cramer–Shoup encryption scheme, 340–342
Cramer–Shoup signature scheme, 339, 340
crib, 58
CRL, 258
cross-certification, 257
CRT, 13, 14
CTR Mode, 135, 138
cycle, 399
cyclic group, 5, 403
CZK, 372

Daemen, 131
Damgård, 376
data integrity, 160, 161
Davies–Meyer hash, 159
DDH, 170, 171, 292, 293, 308, 309, 314, 320, 341, 342
DEA, 126
decipherment, 37
decision Diffie–Hellman, 170, 171, 292
decision problem, 299, 300
decryption, 37
DEM, 160, 327, 329–332
DES, 47, 78, 123–129, 131, 132, 134, 141, 161, 163, 260
DHAES, 333
DHIES, 322, 331–334, 365
difference, 395
differential cryptanalysis, 124, 126
Diffie, 167, 217, 219
Diffie–Hellman Key Exchange, 217
Diffie–Hellman problem, 170, 171, 179, 218, 292, 341
Diffie–Hellman protocol, 218, 219, 229, 230, 260
digital certificate, 256–259, 261
digital signature, 220
Digital Signature Algorithm, 222
Digital Signature Standard, 222
dimension, 415
discrete logarithm, 5
discrete logarithm problem, 169, 170, 178, 192, 201–214, 217, 222, 224
discriminant, 24, 275
disjoint, 399
disjunctive normal form, 349
distributive, 4, 410, 411
DLOG, 229
domain, 397
domain parameters, 178, 218, 223, 340

DSA, 217, 222–227, 229, 231, 233, 238, 249, 259, 264, 265, 282, 315–318, 322, 338
DSS, 222
Durfee, 281, 282

EC-DH, 218, 219
EC-DSA, 222, 224, 225, 229, 231, 264, 315, 317, 318
ECB Mode, 134–136, 138, 139, 221
ECPP, 187, 310
ElGamal, 365
ElGamal encryption, 78, 167, 178, 179, 183, 223, 233, 287, 291–293, 297, 320–322, 325, 333, 341, 342, 364
elliptic curve, 23–33, 169, 178, 183, 187, 213, 214, 218, 219, 222, 224–226, 229, 237, 239, 245
Elliptic Curve Method, 168
Ellis, 167
encipherment, 37
encryption, 37
Enigma, 49–54, 56, 57, 60–62, 64–66, 69, 70, 72, 96
entropy, 84–88, 142
equivalence class, 396
equivalence relation, 23, 25, 395, 396, 409
error-correcting code, 351
Euclidean algorithm, 10–12, 246, 281
Euclidean division, 11, 241, 404
Euler ϕ function, 6, 404, 410
exhaustive search, 94
existential forgery, 296, 313
extended Euclidean algorithm, 12, 13, 31, 169, 172, 173

factor group, 406
factorbase, 192
factoring, 168, 169, 171, 222
feedback shift register, 109
Feistel, 323
Feistel cipher, 125, 126, 131
female, 62, 63
Fermat test, 185, 186, 310
Fermat's Little Theorem, 7, 184, 190
Fiat–Shamir heuristic, 374
field, 6, 131, 411
Filter generator, 117
finite field, 7–9, 23, 28–31, 111, 169, 178, 183, 187, 203, 205, 212, 213, 218, 222, 226, 229, 237, 245, 246, 249, 411
finite-dimensional, 413
fixed field, 9
flexible RSA problem, 338
Floyd's cycle finding algorithm, 207–209
forking lemma, 315
forward secrecy, 217
Franklin, 265
Franklin–Reiter attack, 280, 281
frequency analysis, 40
Frobenius endomorphism, 29
Frobenius map, 9, 29
Fujisaki, 325, 331

full domain hash, 318
function, 397

Galois field, 9, 411
Gap-Diffie–Hellman problem, 334
garbled circuit, 384–387
Gauss, 183
gcd, 5, 11, 12
GCHQ, 167
Geffe generator, 115–117
generator, 5, 403
Gennaro, 338
GHR signature scheme, 338, 339
Gillogly, 72
Goldwasser, 187, 294, 295, 338
Goldwasser–Micali encryption, 294, 295
Gram–Schmidt, 274–277
graph, 299
graph isomorphism, 369, 370, 372
greatest common divisor, 5, 8, 10, 45
group, 4, 184, 201, 402–409
GSM, 118

Halevi, 338
Hamming weight, 234
hard predicate, 306
hash function, 153–161, 217, 221, 222, 226, 228, 291, 296, 314–324, 374
Hasse's Theorem, 213
Hastad, 280
Hastad's attack, 280, 281
HDH, 332
Hellman, 167, 201, 217, 219, 302
hiding, 362
HMAC, 161
homomorphic property, 292
homomorphism, 202, 407–409
honest-but-curious, 383, 384, 391
honest-verifier, 372, 373
Howgrave-Graham, 277
Huang, 187
hybrid cipher, 327, 330, 331
hyperelliptic curve, 187

IDEA, 259
ideal, 196, 197, 410
identity, 401, 402
identity based encryption, 265
identity based signature, 265
identity certificate, 256
image, 408
independent, 20
index, 406
index of Coincidence, 72
index-calculus, 212
indian exponentiation, 235
indistinguishability of encryptions, 288, 291, 328
information theoretic security, 78, 288, 362
injective, 81, 292, 398, 409
inner product, 273
intersection, 395
inverse, 398, 401, 402
IP, 371, 372
irreducible, 8
ISO, 126
isomorphic, 8, 409
isomorphism, 24, 409, 411

Jacobi symbol, 16
Jensen's inequality, 85
joint entropy, 86
joint probability, 19

Karatsuba multiplication, 241, 248, 249
Kasiski test, 45
KASUMI, 118
KEM, 327, 330–332
Kerberos, 147, 148
Kerckhoffs' principle, 93
kernel, 408
key agreement, 144, 150, 151, 256
key distribution, 83, 94
key equivocation, 87
key escrow, 263
key exchange, 142, 218
key ring, 260
key schedule, 124, 129
key transport, 144, 217, 218, 229, 260, 330, 333
Kilian, 187
knapsack problem, 299–305

Lagrange interpolation, 352, 353, 356, 359, 389
Lagrange's Theorem, 7, 184, 246, 405, 406
Las Vegas Algorithm, 175
Las Vegas algorithm, 309
lattice, 271, 273–279, 304, 305
Legendre symbol, 15, 16, 198, 295
Lehmer, 187
Lenstra, 276
LFSR, *109–111*, *114–119*
linear combination, 413
linear complexity, 96, 114–116
linear complexity profile, 115
linear feedback shift register, 109
linear independence, 413
linearly dependent, 413
linearly independent, 274, 413
LISP, 261
LLL algorithm, 276, 277, 279, 305
LLL reduced basis, 276
long Weierstrass form, 23
Lorenz cipher, 96–107
Lovász, 276
Lucifer, 126
lunch-time attack, 289

MAC, 160–163, 321, 327, 329, 333
malleable encryption, 290
man in the middle attack, 219
manipulation detection code, 153

MARS, 124
Matyas–Meyer–Oseas hash, 159
MD4, 156–158
MD5, 156, 259, 291
MDC, 153
Menezes, 229
Merkle, 302
Merkle–Damgård, 155, 160, 161
Merkle–Hellman, 303, 304
message authentication code, 153, 160, 163, 220
message recovery, 220, 221
Micali, 294, 295, 338
Miller–Rabin test, 186, 187, 310
millionaires problem, 383
Miyaguchi–Preneel hash, 159
mode of operation, 123, 136
modular arithmetic, 396
modulus, 3
monotone structure, 348
Monte-Carlo algorithm, 309
Montgomery, 239
Montgomery arithmetic, 243–246
Montgomery multiplication, 245
Montgomery reduction, 244, 245
Montgomery representation, 243, 244
MQV protocol, 229, 230
multi-party computation, 383–392

natural language, 88
Needham, 148
Needham–Schroeder protocol, 145–148
negligible, 328
Nguyen, 198
NIST, 124
non-deterministic polynomial time, 300
non-negligible probability, 313
non-repudiation, 220
nonce, 143, 146–148, 150, 230, 377
normal, 405
normal subgroup, 405, 406
NP, 300–302, 309, 310, 371
NP-complete, 301, 302
NSA, 126
Number Field Sieve, 169, 188, 213, 224
Nyberg–Rueppel signature, 228

OAEP, 322, 323
oblivious transfer, 365, 366, 387
OFB Mode, 135, 137, 138
Okamoto, 325, 331
one-time pad, 78, 83, 94
one-way function, 168
operation, 400
optimal normal bases, 245
Optimized Asymmetric Encryption Padding, 322
order, 403
orthogonal, 273, 274
Otway–Rees protocol, 146, 147

partial order relation, 396
passive attack, 91, 94, 289, 294, 328
Pedersen commitment, 363, 375, 377
Pederson commitment, 364
pentanomial, 246
perfect cipher, 90
perfect security, 78, 80, 287, 288, 313
perfect zero-knowledge, 371
period of a sequence, 110
permutation, 47, 48, 50–52, 55, 59, 65, 398–401, 403
permutation cipher, 47
PGP, 259, 260
PKI, 259
plaintext, 37
plaintext aware, 291, 325
plugboard, 52–57, 61, 62, 64, 67–71, 74
Pocklington, 187
Pohlig, 201
Pohlig–Hellman algorithm, 201, 204, 205
point at infinity, 23
point compression, 32
point multiplication, 237
Pointcheval, 315, 325
Pollard, 190, 206, 207
Pollard's Kangaroo method, 209
Pollard's Lambda method, 209
Pollard's Rho method, 169, 206, 208, 213
polynomial security, 287–293
predicate, 306
preimage resistant, 153
primality proving algorithm, 183
prime ideal, 196, 197
prime number, 404
Prime Number Theorem, 183
primitive, 110
primitive element, 9
principal ideal, 410
private key, 93
probabilistic signature scheme, 319
probability, 19
probability distribution, 19
probable prime, 185
product group, 409
Project Athena, 147
projective plane, 23
projective point, 23, 24
proof of primality, 187
provable security, 313
PRSS, 357, 358, 392
PRZS, 358, 392
pseudo-prime, 185
pseudo-squares, 18, 294
pseudorandom secret sharing, 357, 358, 392
pseudorandom zero sharing, 358, 392
PSPACE, 371, 372
public key, 93
public key certificate, 260
public key cryptography, 93, 142, 167, 168
public key cryptosystems, 37
Public Key Infrastructure, 259

public key signatures, 219

Qu, 229
quadratic reciprocity, 15
Quadratic Sieve, 168, 188, 195
QUADRES, 169, 172, 294, 295, 314, 320
qualifying set, 347–350
quotient group, 406

RA, 258
Rabin, 167, 180, 183, 338
Rabin encryption, 180, 181
random oracle model, 291, 314–320, 322, 323, 337–340, 366, 373
random self-reduction, 308
random variable, 19
random walk, 206, 207, 209–211
RC4, 119, 120, 260
RC5, 119, 124, 260
RC6, 119, 124
redundancy, 89
redundancy function, 228
Reed–Solomon code, 351–356, 392
Reed-Solomon code, 357
reflector, 50, 52, 57, 60
reflexive, 396
registration authority, 258
Rejewski, 56
relation, 395, 409
relatively prime, 6
replicated secret sharing, 350
residue classes, 396
Rijmen, 131
Rijndael, 47, 78, 123–125, 131–134, 141, 245
ring, 5, 131, 410, 411, 413
RIPEMD-160, 156
Rivest, 119, 219, 338
Rogaway, 322, 333
round function, 124
round key, 124
Rozycki, 56
RP, 309, 310
RSA, 13, 78, 167–169, 172–178, 180, 181, 183, 191, 192, 213, 214, 217, 220–224, 228, 229, 231, 233, 234, 238, 239, 241, 249, 259, 260, 264–266, 271–273, 277, 278, 280–284, 287, 291–294, 296, 297, 302, 306–308, 313, 314, 318–320, 322–324, 327, 331, 332, 337–340, 342, 358, 359
RSA-FDH, 318, 319
RSA-KEM, 331, 332
RSA-OAEP, 320, 322–325, 331, 332
RSA-PSS, 319, 338

S-Box, 127–129, 132, 134
S-expressions, 261
safe primes, 191, 338
Schmidt, 57
Schnorr signature, 374
Schnorr signatures, 226, 227, 315–318, 373, 374
Schoenmakers, 376
Seberry, 320–322, 331, 332
second preimage resistant, 154
secret key cryptosystems, 37
secret sharing, 143, 263, 347–351, 356–358, 384, 388–392
Secure Socket Layer, 260
security parameter, 313
selective forgery, 296
semantic security, 287, 288, 291, 313, 320, 328, 331
Serpent, 124
session key, 142
SHA, 158
SHA-0, 158
SHA-1, 156, 158, 159, 259, 291, 339
SHA-2, 156
SHA-256, 156
SHA-384, 156
SHA-512, 156
Shamir, 219, 265, 351
Shamir secret sharing, 351, 356–359, 379, 384, 389–392
Shamir's trick, 238, 239
Shanks, 204
Shanks' Algorithm, 16, 171
Shannon, 81, 84
Shannon's Theorem, 81, 83, 84
Sherlock Holmes, 41
shift cipher, 39, 41, 44, 45, 78, 82, 83, 88
short Weierstrass form, 25
Shoup, 330, 331, 335, 339, 340
Shrinking generator, 118
Sigma protocol, 373–377
signature, 220–222
signature scheme, 222
signed sliding window method, 237
signed window method, 237
Sinkov statistic, 54, 74
sliding window method, 236, 237
small inverse problem, 281, 282
smart card, 256
Smith Normal Form, 192
smooth number, 188
smoothness bound, 192
soundness, 371, 373
span, 413
special-soundness, 373, 375, 376
SPKI, 147, 261, 262
spurious keys, 88, 90
square and multiply, 235
SSH, 260
SSL, 260
standard basis, 414
Stern, 315, 325
stream cipher, 39, 41, 44, 95, 96, 114, 115, 119, 123, 125, 135, 137
strong RSA, 338, 339
subfield, 9
subgroup, 404, 405
substitution cipher, 41, 44, 49, 78, 88, 90

super-increasing knapsack, 303
supersingular, 29, 214
surjective, 398, 409
symmetric, 396, 409
symmetric cryptosystems, 37
symmetric key, 93

TCP, 260
threshold access structure, 348, 351
Tiltman, 102
timestamp, 144, 147, 150
total order relation, 396
trace of Frobenius, 28
traffic analysis, 94
transitive, 396, 409
trapdoor one-way function, 168
trial division, 168, 184, 188
trigrams, 38
trinomial, 246
Triple DES, 126, 260
trusted third party, 143, 256
TTP, 143, 256
Turing, 64, 66, 67
Tutte, 102
Twofish, 124

UMTS, 118
unconditionally secure, 78
unicity distance, 88, 90
union, 395
universal one-way hash function, 321

Vanstone, 229
vector space, 412–414
vector subspace, 412
Vernam cipher, 83, 95
Vigenère cipher, 44, 78

Weierstrass equation, 23
Welchmann, 66
Wide-Mouth Frog protocol, 144, 150
Wiener, 271
Wiener's attack, 272, 277, 281, 282
Williamson, 167
window method, 235–237
witness, 185

X509, 260–262

Yao, 384, 385, 387
Yao circuit, 384

zero-knowledge, 227, 318, 369–378
Zheng, 320–322, 331, 332
ZPP, 310
Zygalski, 56, 62
Zygalski sheet, 62, 63